The Pragmatic Programmers

现代数据库和
NoSQL运动指南

七周七数据库

Seven Databases in Seven Weeks

A Guide to Modern Databases and the NoSQL Movement

[美] Eric Redmond Jim R. Wilson 著

王海鹏 田思源 王晨 译



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

七周七数据库

Seven Databases in Seven Weeks

A Guide to Modern Databases and the NoSQL Movement

[美] Eric Redmond Jim R. Wilson 著

王海鹏 田思源 王晨 译



人 民 邮 电 出 版 社

北 京

图书在版编目（ＣＩＰ）数据

七周七数据库 / (美) 雷德蒙 (Redmond,E.) , (美) 威尔逊 (Wilson,J.R.) 著 ; 王海鹏, 田思源, 王晨译. -- 北京 : 人民邮电出版社, 2013.7
ISBN 978-7-115-31224-2

Ⅰ. ①七… Ⅱ. ①雷… ②威… ③王… ④田… ⑤王… Ⅲ. ①数据库系统 Ⅳ. ①TP311.13

中国版本图书馆CIP数据核字(2013)第042507号

版权声明

◆ 著　　[美] Eric Redmond
　　　　Jim R.Wilson
译　　王海鹏　田思源　王　晨
责任编辑　陈冀康
责任印制　程彦红　杨林杰

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000　1/16
印张：20.5
字数：461 千字　2013 年 7 月第 1 版
印数：1 – 3 500 册　2013 年 7 月北京第 1 次印刷

著作权合同登记号　图字：01-2012-4612 号

定价：59.00 元

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154

内容提要

如今，我们要面对和使用的数据正在变得越来越庞大和复杂。如果说数据是新的石油。那么数据库就是油田、炼油厂、钻井和油泵。作为一名现代的软件开发者，我们需要了解数据管理的新领域，既包括 RDBMS，也包括 NoSQL。

本书遵循《七周七语言》的写作风格和体例，带领你学习和了解当令最热门的开源数据库。在简单的介绍之后，本书分章介绍了 7 种数据库。这些数据库分别属于 5 种不同的数据库风格，但每种数据库都有自己保存数据和看待世界的方式。它们依次是 PostgreSQL、Riak、Apache HBase、MongoDB、Apache CouchDB、Neo4J 和 Redis。本书将深入每一种数据库，介绍它们的优势和不足，以及如何选取一种最符合你的应用需求的数据库。

本书适合数据库架构师、数据库管理员，以及想要了解和学习各种 NoSQL 数据库技术的程序员阅读。本书将帮助读者了解、选择和应用这些数据库，从而更好地发挥日益增长的大数据的能力。

译者简介

王海鹏，1994 年毕业于华东师范大学。软件开发者，独立的咨询顾问、培训讲师、译者。拥有 20 年编程经验，已翻译二十余本软件开发书籍。目前主要感兴趣的领域是软件架构和方法学，致力于提高软件开发的品质和效率。新浪微博：@王海鹏 Seal。

田思源，资深 IT 人士，从事软件、互联网行业多年，有多部译作面世。现居北京。新浪微博：@胡试之。

王晨，IBM 中国系统与科技研发中心的软件工程师，从事硬件集成管理系统的开发测试工作。新浪微博：@wwwwwch。

特别感谢李元佳（第 2 章）、谢磊（第 4 章）、程显峰（第 5 章）、李子骅（第 8 章）参与审阅译稿。

序

在科罗拉多州布雷肯里奇（Breckenridge）的滑雪季里，登上 Beaver 雪道运行超级缆车，一眼望去，滑雪道的斜坡被精心地修整过，而山上的植被和覆盖的雪层却依然如旧。我们在心里发问，新雪在哪里呢？没有新雪，滑雪的体验就不那么美妙了。

1994 年，作为 IBM 在奥斯汀的数据库开发实验室工作的雇员，我的感觉非常类似。当时我刚在奥斯汀的德克萨斯大学学习了面向对象数据库，因为在关系数据库主宰了 10 年之后，我想面向对象数据库真的有机会深入人心。但是，接下来的 10 年和之前一样，出现了更多同样的关系模型。我沮丧地关注着 Oracle、IBM 和其他以 MySQL 为首的开源解决方案，它们强劲伸展着的枝叶，完全挡住了阳光，妨碍了肥沃的土壤上正在萌芽的其他解决方案。

随着时间的推移，用户界面从绿屏幕变成了客户端-服务器的方式，又变成了基于互联网的应用，但关系层的代码同样张开无情的铁幕，几十年如一日地称职而单调。所以，我们期待着一场新雪。

然后新雪终于降临了。起初，雪花甚至不足以掩盖早行者的足迹，但暴雪随后到来，覆盖大地，带来了完美的滑雪体验，这正是我们渴望的不同和品质。在过去的一年里，我醒过来时发现，数据库的世界也覆盖了一层新雪。当然，关系数据库还在，你可以从开源 RDBMS 软件中获得令人吃惊的丰富体验。你可以创建集群，进行全文搜索，甚至进行模糊搜索。但你不再受限于某种方式。一年里我没有创建过一个完整的关系型解决方案。在这段时间里，我使用了一个文档数据库和几种键-值数据库。

真实情况是，关系数据库在灵活性和可伸缩性方面不再处于垄断地位。对于我们要构建的各类应用，还有更多合适的模型，更简单、更快速、更可靠。作为在 IBM 的奥斯汀实验室待了 10 年，与同事和客户从事数据库工作的人，我被这种进步惊呆了。在本书中，你会看到一些例子，完美地覆盖了数据库领域最重要的进展，正是这些数据库支撑了互联网的发展。在键-值存储库中，你会看到伸缩性极好、极为可靠的 Riak，还会看到 Redis 中漂亮的查询机制。在列型数据库社区，你将体验到 HBase 的威力，它是关系数据库模型的近亲。在文档数据库中，你会看到伸缩性极好的、优雅的解决方案，处理深层嵌套的文档。你还会看到 Neo4j 在图形数据库上的应用，支持快速地在关系上导航。

要成为更好的程序员或数据库管理员，你不必使用所有这些数据库。随着 Eric Redmond

和 Jim Wilson 引导你走过的这段神奇的旅程，每一步都会让你更聪明，所获得的深刻见解对于软件职业来说是无价的。你会知道每种平台的闪光之处和最大局限。你会看到行业的发展方向，理解行业的驱动力。

享受这段旅程吧。

Bruce Tate，《七周七语言》的作者

2012 年 2 月

德克萨斯州奥斯汀

作者访谈

Q：你们怎么选择这七种数据库的？

Eric: 我们确实有一些选择标准，但没有明确列出来。这些数据库必须是开源的，因为我们不想介绍让读者绑定某公司的数据库。对于 5 种数据库类型（关系型、键-值对型、列型、文档型、图型），每种至少需要一个实现。然后我们选择一些数据库，它们能够用实例展示我们想介绍的一些一般概念，如 CAP 原理或 MapReduce。最后，我们选择一些彼此是很好的竞争对手的数据库。所以我们选择了 MongoDB 和 CouchDB（二者实现文档数据库的不同方式）。我们选择 Riak 是因为它是 Dynamo（亚马逊的数据库）的一种实现，可以与 HBase 进行比较，而后者是 BigTable（谷歌的数据库）的一种实现。

Jim: 我们这本书的主要目标是向读者介绍现有的选择。我们的选择基本上服务于这个目标。即便如此，这也是一个相当长的迭代过程。我们知道，不论我们选了哪些数据库，都会有人问，为什么我们选择或没选择他们喜欢的产品。这首先取决于我们想讨论的数据库类型，然后选择的数据库既要代表某个类型，又要相对比较受欢迎。

例如，我们选择了 PostgreSQL，因为它几乎严格地遵守 SQL 标准，同时又不如 MySQL 这样的开源竞争产品那样知名。类似地，虽然 HBase 和 Cassandra 都是面向列的数据库，但我们选择了 HBase，因为 Cassandra 是混合类型，它同时包含了来自 BigTable 论文和 Dynamo 论文的思想。

Q：数据库正在快速变化。现在你们希望当初选了哪些？

Eric: 有几百种数据库可以选择，但我们很高兴地看到，一年之后，我们的选择仍然在变得更强。但是，如果再来一次，我会加入 Triplestore（如 Mulgara），因为语义网正让这种数据存储方法逐渐变得热门。我也会在 Neo4j 的 Cypher 语言上花更多的时间，或更详细地介绍 Hadoop，因为分析是数据存储的一大部分工作。

Jim: 是的，数据库在快速地变化，这体现在两个方面。首先，可用数据存储技术的领域在近年来有了爆炸式增长。越来越多的不同数据库正在涌现，填补不同的小众需求。在另一方面，各种数据库本身也在快速发展。即使在小版本之间，现代的 NoSQL 数据库也在加入越来越多的特征，目的是占领更多的市场，保持竞争力。因此，也产生了一些趋同效应，这使得选择更为困难，因为更多的产品可以满足你的需求。

我还是认为，我们选择的 5 种类型和 7 种数据库满足了我们设定的条件。但我还想写其他的数据库。包括一些长期受欢迎的数据库，如 SQLite，以及一些你可能想不到的数据库，如 OpenLDAP 或 SOLR（一个倒排索引/查找引擎）。

Q：为什么你们决定要写这本书？

Eric: Jim 和我很早就在讨论写这样一本书了。大约一年半以前，他发了一封只有标题的电子邮件: "七周七数据库？" 这个标题打动了我。我们都喜欢 Bruce 的《七周七语言》，这似乎是探讨这个新兴领域的完美方式。

Jim: 早在 2010 年 3 月，Eric 和我就对写本 NoSQL 的书进行了探讨。那时候，关于这个术语有许多讨论，但也有许多迷惑。我们认为我们可以为讨论带来某种结构，并将所有最新进展告诉不那么了解最新情况的人。

在读了 Bruce A. Tate 的《七周七语言》之后，我想，"七种数据库怎么样？" Eric 提交了一份建议，几周后我们就开始动手写了。

Q：你对目前和将来的数据库怎么看？

Eric: 我是 Neo4j 的粉丝。我们在本书中介绍了它，但老实说，我们选它是因为想探讨一个开源的图数据库。但在过去的一年里，它确实非常成功。我相信，今年会有更多的人采用图数据库。

在我们没有介绍的数据库中，我认为 ElasticSearch 很清楚获得了支持。OrientDB 也很有趣，因为它可以作为关系型、键值对型、文档型或图数据库。我认为将来你会看到更多这样的多类型表现。就像我们前面提到的，Triplestores 也得到一些支持，虽然他们的问题集与一般的图型数据库有很大的交集。

Jim: 当然，有很多数据库，但至少有两种是我个人希望仔细研究的: ElasticSearch 和 doozer。

ElasticSearch 是分布式的、对等的、支持 REST/JSON 的文档搜索引擎。ElasticSearch 采用分布式的 Lucene 索引作为核心，允许 REST 客户端根据模糊的条件查找文档。每个人都需要一个搜索引擎，ElasticSearch 让这一点变得容易了。

Doozer 是一个快速的、无头一致（headless consensus）的引擎。它是 Heroku 的一群聪明人用 Go 语言写的。Dozzer 对于存放小块的重要信息是很好的，这些信息绝对需要一致（如集群的配置元数据），但没有单点失效。

前　言

如果说数据是新的石油，那么数据库就是油田、炼油厂、钻井和油泵。数据存放在数据库中，如果你有兴趣利用它，那么掌握相应的现代化的工具就是好的开始。

数据库是工具，它们是到达终点的手段。每种数据库都有自己保存数据和看待世界的方式。你对它们的理解越多，就越能随心所欲，在日益增长的大数据上，就能更好地利用它们潜在的能力。

为什么是 7 种数据库

早在 2010 年 3 月，我们就想写本关于 NoSQL 的书。NoSQL 这个术语已经聚集了人气，虽然许多人都在谈论，但关于它似乎也存在相当多的困惑。NoSQL 到底意味着什么？包含哪些类型的系统？对于开发优秀的软件，它将产生怎样的影响？这些就是我们想要回答的问题，既是为我们自己，也是为别人。

在读了 Bruce Tate 的典范性的著作《七周七语言》后，我们知道他做得很对。他循序渐进地介绍语言的方式引起了我们的共鸣。我们觉得用同样的方式讲授数据库，将会提供一个很好的环境，回答这些棘手的 NoSQL 问题。

本书内容

本书针对的是有经验的开发者，他们希望全面地理解现代数据库的整体情况。本书不要求读者以前在数据库方面有经验，但有数据库经验会有助于学习本书。

在简单的介绍之后，本书分章介绍了 7 个数据库。这些数据库分属 5 种不同的数据库类型或风格，这在第 1 章中有介绍。它们依次是 PostgreSQL、Riak、Apache HBase、MongoDB、Apache CouchDB、Neo4j 和 Redis。

每章都设计成一个长周末的学习量，分为三天。每天结束时都有一些练习，扩展刚刚介绍的主题和概念。每章最后都有一段总结性的讨论，总结了这种数据库的优点和缺点。你既可以学得快一点，也可以学得慢一点，但重要的是先掌握每天的概念，再继续后面的学习。我们试着设计了一些例子来探索每种数据库的独特之处。要真正理解这些数据库提

供的能力，必须花些时间来使用它们，这意味着要动手实践。

虽然你可能想跳过某些章，但我们设想你是按章节顺序阅读这本书的。某些概念，如映射-归约（mapreduce），在前面的章节中深入地进行了介绍，所以在后面的章节中就略过了。本书旨在实现对现代数据库的一致理解，所以建议你完整地阅读本书。

不包含的内容

在阅读本书之前，你应该知道它不包含哪些内容。

本书不是安装指南

安装本书中提到的数据库有时候容易，有时候有些挑战，有时候非常棘手。对于某些数据库，可以使用提供的安装包；而对于另一些数据库，需要编译源代码。我们会不时提供一些有用的提示，但主要还是靠你自己。省略安装步骤让我们能安排更多有用的例子和概念讨论，这才是你真正想要的，对吗？

本书也不是管理手册

出于对安装同样的考虑，本书也不会介绍管理手册里的所有内容。每种数据库都有大量的选项、设置、开关和配置细节，绝大部分都能在 Web 上找到详尽的文档。我们更关心介绍有用的概念，完全深入进去，而不是仅关注日常操作。虽然数据库的一些特点会根据操作设置而改变（我们可能会讨论这些特点），但由于篇幅有限，我们不可能介绍所有可能配置的全部具体细节。

对Windows用户的说明

本书本身就讨论选择，主要是针对*nix平台上的开源软件。微软的环境作为集成环境有点困难，它限制了许多选择，只留下一个较小的、预定义的子集。因此，我们介绍的数据库是开源的，由*nix系统的用户开发（也主要为他们服务）。这不是我们的偏见，只是当前真实情况的反映。所以，我们假定教程式的例子运行在*nix的shell下。如果你运行Windows并希望给它一个尝试的机会，我们推荐安装Cygwin[1]，这样更容易成功。你也可以考虑运行一个Linux虚拟机。

---

[1] http://www.cygwin.com/

代码示例和惯例

本书包含各种语言的代码。部分原因是因为我们介绍的这些数据库本身使用语言不同。我们曾试着将语言局限在 Ruby/JRuby 和 JavaScript。我们更喜欢命令行工具，而不是脚本语言，但我们会引入其他一些语言来完成工作，如 PL/pgSQL（Postgres）和 Gremlin/Groovy（Neo4j）。我们也会尝试使用 Node.js，编写一些服务器端的 JavaScript 应用。

除非特别注明，代码清单都是完整的，通常可以直接执行。根据所涉及的语言规则，突出了示例和代码片段中的语法。shell 命令以$开始。

在线资源

本书的Pragmatic Bookshelf页面[1]是很好的资源。从上面可以下载本书中的所有源代码。你也会找到一些反馈工具，如社区论坛和勘误提交，你可以通过它们对本书未来的版本提出改进建议。

感谢你陪伴我们完成现代数据库的观光之旅。

Eric Redmond 和 Jim R. Wilson

---

1 http://pragprog.com/book/rwdata/seven-databases-in-seven-weeks

致　谢

本书内容涉及范围都比较宽泛，只靠两位作者是无法完成的。它需要许多非常聪明的人的努力，他们有超人般的眼睛，能尽可能多地发现错误，针对这些技术的细节提供有价值的见解。

我们要感谢所有贡献出时间和专业知识的人（不分先后）：Ian Dees、Mark Phillips、Jan Lenhardt、Robert Stam、Oleg Bartunov、Dave Purrington、Daniel Bretoi、Matt Adams、Sean Copenhaver、Loren Sands-Ramshaw、Emil Eifrem 和 Andreas Kollegger。最后，还要感谢 Bruce Tate 的经验和指导。

我们还要真诚地感谢 Pragmatic Bookshelf 的整个团队。感谢他们提出这个大胆的项目，并看着我们完成。我们特别要感谢编辑 Jackie Carter。是你的耐心反馈，才有了今天这本书。感谢整个团队辛勤的工作，找出了我们所有的错误。

最后但同样重要的是，感谢 Frederic Dumont、Matthew Flower、Rebecca Skinner 和所有严格的读者，如果没有你们的学习热情，我们就不会有机会提供服务了。

对于这里遗漏的人，我们希望你接受道歉。我们肯定不是有意的。

Eric 想说：亲爱的 Neolle，你不是特别的，你是唯一的，这要好得多。谢谢你又忍受我完成了一本书。感谢数据库的创建者和贡献者，为我们提供了写书的内容和谋生的工具。

Jim 想说：首先，我要感谢我的家庭。Ruthy，你无限的耐心和鼓励温暖了我的心。Emma 和 Jimmy，你们是两个聪明鬼，爸爸永远爱你们。还要感谢所有无名英雄，他们盯着 IRC、消息板、邮件列表和 bug 管理系统，时刻准备帮助需要的人。你们对开源的贡献让这些项目一直激动人心。

目　录

第 1 章

概述

当前是数据库世界的一个重要时刻。多年来，无论针对的问题是大还是小，关系模型一直是事实上的选择。我们不指望关系数据库会很快消失，但是人们正在从 RDBMS 的迷雾中走出来，寻找替代的方案，如无模式或可替代的数据结构，可简单复制，具有高可用性，可横向扩展，以及新的查询方法。这些选择统称为 NoSQL，而 NoSQL 占据了本书的大部分内容。

本书将探讨七种数据库，涉及各种数据库风格。在阅读本书的过程中，你将了解每个数据库具有的各种功能和折中，如持久性与速度、绝对一致性与最终一致性等，并学会如何针对你的使用场景，做出最好的决策。

1.1　从一个问题开始

本书的核心问题是：哪种数据库或数据库组合最好地解决了你的问题？读完本书，如果你知道如何根据特定需求和手头的资源做出这种选择，我们会很高兴。

但要回答这个问题，你需要了解你的选择。为此，我们将带你深入这 7 个数据库，揭示精华，并指出瑕疵。你将亲手尝试 CRUD 操作，发挥你使用的模式的力量，并找到下面这些问题的答案：

- 这是什么类型的数据库？数据库分为各种类型，例如，关系型、键-值型、多列型、面向文档型和图型。流行的数据库（包括本书中介绍的）一般可以划分为这几大类型。你将了解每种类型的数据库，以及它们最适合的各种问题。我们对本书涉及的数据库精心挑选，以覆盖这些类型，包括一个关系数据库（Postgres），两个键-值存储数据库（Riak 和 Redis），一个面向列的数据库（HBase），两个面向文档的数据库（MongoDB 和 CouchDB），以及一个图数据库（Neo4j）。

- *驱动力是什么？*数据库不是凭空产生的。它们是为了解决实际使用中提出的问题。在 RDBMS（关系数据库管理系统）出现的环境中，数据库查询的灵活性比灵活的模式更重要。另一方面，建立面向列的数据库是为了适于存储跨多机的大量数据，而数据之间的关系退居次要地位。我们将介绍使用每个数据库的场景和相关的例子。

- *如何与数据库交互？*数据库往往支持多种连接选项。只要某个数据库有交互式的命令行界面，我们会首先使用它，之后再介绍其他方法。如果需要编程，我们主要使用 Ruby 和 JavaScript，尽管不时会用到其他几种语言，如 PL/pgSQL（Postgres）和 Gremlin（Neo4j）。更深入一层，我们将讨论诸如 REST（CouchDB 和 Riak）和 Thrift（HBase）协议。第 9 章将介绍一个更复杂的数据库环境，它由 Node.js JavaScript 实现联接在一起。

- *每种数据库的独特性体现在哪里？*任何数据存储库都支持写入和读回数据。它们在其他的方面彼此大不相同。有些数据库允许对任意字段的查询。有些数据库提供快速索引查找。有些数据库支持自由定义的查询（ad hoc query）；而其他的数据库的查询必须先规划。模式是数据库所强制的一个刚性框架，或仅仅是一些随意商定的准则？理解每种数据库的功能和限制，将有助于挑选适合你的工作的数据库。

- *每种数据库的性能如何？*这个数据库如何工作？其开销如何？它支持分片吗？复制呢？它是否使用一致散列均匀地分布数据？它将相似的数据放在一起吗？这个数据库为读、写或其他操作做了优化吗？如果你能对优化进行控制，程度如何？

- *每种数据库的可伸缩性如何？*可伸缩性与性能相关。没有上下文，谈论可伸缩性一般不会有结论。本书会提供背景知识，你在建立这个上下文时就能提出正确的问题。虽然如何扩展每个数据库的讨论会有意淡化，但在这些章节里，你会发现每种数据存储库是更容易实现横向扩展（MongoDB、Hbase 和 Riak），还是传统的纵向扩展（Postgres、Neo4j 和 Redis），或者介于两者之间。

我们的目标不是将某种数据库的新手培养成大师。如果这样做的话，其中任何一个数据库都将充满整本书的篇幅。但最终你应该能够牢牢把握每个数据库的优势，以及它们的不同。

1.2 风格

如同音乐一样，各种数据库有着其本身独特的风格。所有的歌曲都使用同样的音符，但是有些音符对某些网络的歌曲更合适。少有人驾驶着敞篷车，沿着 405 号公路超速行驶，同时播放巴赫的《B 小调弥撒曲》。同样，在某些情况下，一些数据库比其他数据库更好些。你要问的不是“我能够用这个数据库来存储和完善数据吗？”而是“我应该用这个数据库吗？”

本节将要探讨 5 种主要的数据库类型，也会简单介绍每种类型中我们要关注的数据库。

重要的是要记住，你将面临的大多数数据问题，可以用本书中的大部分或全部数据库解决，更别说还有其他数据库。问题不是某种数据库风格是否能别生搬硬套地用来为你的数据建模，而是它是否最适合你的问题领域、使用模式，以及可用的资源。你将学会预测一种数据库是否在本质上对你有用，而这是一门艺术。

1.2.1 关系数据库

关系模型通常是大多数有数据库经验的人首先想到的。关系数据库管理系统（Relational DataBase Management System，RDBMS）是以集合理论为基础的系统，实现为具有行和列的二维表。与 RDBMS 交互的标准方法，是用结构化查询语言（Structured Query Language，SQL）编写查询。数据值具有类型，可以是数字、字符串、日期、未解释的二进制大对象，或其他类型。系统强制使用类型。重要的是，表可以联接并转化为新的、更复杂的表，因为它们的数学基础是关系（集合）理论。

有许多开源关系数据库可供选择，包括 MySQL、H2、HSQLDB、SQLite 等。第 2 章将介绍 PostgreSQL。

PostgreSQL

PostgreSQL 久经沙场，它是迄今为止我们介绍的最古老和最健壮的数据库。PostgreSQL 符合 SQL 标准，之前曾使用过关系数据库的人都会觉得熟悉它，这为我们将使用的其他数据库提供了一个坚实的比较基础。我们还将探讨一些不大为人熟悉的 SQL 功能以及 Postgres 特有的优势。从 SQL 新手到专家，每个人都能从中有所收获。

1.2.2 键-值数据库

键-值（Key-Value，KV）存储库是我们介绍的最简单的模型。顾名思义，KV 存储库将键与值配对，类似于所有流行编程语言中的映射（或哈希表）。某些 KV 实现允许复杂的值类型，如哈希或列表，但这不是必需的。一些 KV 实现提供了一种迭代遍历键的方式，但这也是额外的好处。如果你将文件的路径视为键而将文件内容作为值，文件系统也可以看成是键-值存储库。因为 KV 存储库对资源的要求非常少，所以这种数据库类型在一些场景中有令人难以置信的高性能，但是当你有复杂的查询和聚合需求时，它一般不会有帮助。

与关系数据库一样，有许多开源的 KV 存储库可以选择。一些较受欢迎的产品包括

memcached（及相关的 memcachedb 和 membase）、Voldemort，以及我们在本书中介绍的两个产品：Redis 和 Riak。

1. Riak

第 3 章介绍的 Riak 不仅仅是一个键-值存储库，它从一开始就支持 HTTP 和 REST 等 Web 方式。它严格实现了亚马逊 Dynamo 的原理，具有一些高级功能，如解决冲突的向量时钟。Riak 中的值可以是任何内容，从纯文本到 XML 到图像数据，而键之间的关系由称为链接（link）的命名结构处理。Riak 是本书中知名度较小的数据库，但它越来越受欢迎，在我们将要讨论的数据库中，它是第一个通过 mapreduce 支持高级查询的数据库。

2. Redis

Redis 提供复杂的数据类型，如有序集合和哈希，以及基本消息模式，如发布-订阅和阻塞队列。它是查询机制最健壮的 KV 存储库之一。在写入磁盘之前先写入内存缓存，Redis 因此获得了惊人的性能，代价是在出现硬件故障的情况下，增加了数据丢失的风险。这一特性使得它适合用于缓存非关键数据，或作为消息代理。我们将它留到最后介绍（参见第 8 章），以便可以用 Redis 与其他数据库配合，构建多数据库应用。

1.2.3 列型数据库

列型（或面向列的数据库）的命名源自于其设计的一个重要方面，即来自一个给定的列（在二维表的意义上）的数据存放在一起。相比之下，面向行的数据库（如 RDBMS），将一行的信息保存在一起。这种差异看起来似乎无关紧要，但实际上这种设计决策的影响很深。在面向列的数据库中，添加列是相当简易的，而且是逐行完成的。每一行可以有一组不同的列，或完全没有，允许表保持稀疏（sparse），而不会产生空值的存储成本。在结构方面，列型数据库大约介于关系数据库和键-值存储库之间。

在列型数据库市场中，竞争相比关系数据库或键-值存储较少。三种最流行的产品是 HBase（在第 4 章中介绍）、Cassandra 和 Hypertable。

HBase

在我们介绍的所有非关系数据库中，这个面向列的数据库与关系模型最为相似。以 Google 的 BigTable 论文作为蓝图，HBase 建立在 Hadoop（一个 mapreduce 引擎）之上，其设计目标是在常用硬件的集群上横向伸缩。HBase 保证了强一致性并提供带行和列的表的功能，这使得 SQL 粉丝有宾至如归的感觉。它直接支持版本控制和压缩，这令它在“大数据”的世界中与众不同。

1.2.4 文档型数据库

当然，面向文档的数据库存储的就是文档。简而言之，文档就像是哈希，具有一个独一无二的标识符（ID）字段和值，值可能是任何类型，包括更多的哈希。文档可以包含嵌套的结构，因此，它们表现出了高度的灵活性，允许有可变域。该系统对输入的数据很少有限制，只要它满足基本要求，可以表示为一个文档。在建索引、自由定义的查询、复制、一致性及其他设计决策等方面，不同的文档数据库采取了不同的方法。需要了解这些差异，及其对特定使用场景的影响，才能在它们之间做出明智地选择。

文档数据库市场中的两大开源产品是 MongoDB（在第 5 章中介绍）和 CouchDB（在第 6 章中介绍）。

1. MongoDB

MongoDB 的设计目标是支持巨大的数据（名字 mongo 是从单词 humongous 中提取的）。Mongo 服务器的配置试图保持一致性：如果你写入了什么内容，随后的读取将得到相同的值（直到下次更新）。这一特性吸引了那些具有 RDBMS 背景的人。它也提供了一些原子读写操作（如递增一个值），以及对嵌套文档结构的深层查询。MongoDB 利用 JavaScript 作为查询语言，支持简单的查询和复杂的 mapreduce 工作。

2. CouchDB

CouchDB 的目标是各种部署场景，从数据中心到桌面，一直到智能手机。CouchDB 是用 Erlang 编写的，具有独特的坚固性，这一点在大部分其他数据库中是缺乏的。由于它的数据文件几乎不可摧毁，即使是面对间歇性的连接丢失或硬件故障，CouchDB 也仍能保持高可用性。像 Mongo 一样，CouchDB 的原生查询语言是 JavaScript。视图包括 mapreduce 函数，它们以文档的形式存储并在节点之间复制，像任何其他的数据一样。

1.2.5 图数据库

这是一种不太常用的数据库类型，图数据库善于处理高度互联的数据。图数据库包含节点及节点之间的关系。节点和关系可以有一些属性（一些键-值对），用来存储数据。图数据库的真正实力是按照关系遍历节点。

第 7 章讨论现在最流行的图数据库 Neo4j。

Neo4j

在遍历自我引用或以其他方式杂乱地连接在一起的数据时，其他数据库常常操作失败。

这正是 Neo4j 使人眼前一亮的地方。使用图数据库的好处在于能够快速在节点和关系之间移动，找到相关数据。图数据库经常用在社交网络应用中，它们因灵活性而受到关注，Neo4j 是其中的佼佼者。

1.2.6　混合使用多种数据库

在实际环境中，各种数据库经常一起使用。使用单一的关系数据库仍然很常见，但随着时间的推移，流行的做法是同时使用几种数据库，利用它们各自的长处，创建一个生态系统，比其各部分的功能总和更强大、更全面、更健壮。这种做法叫做多持久并存（Polyglot Persistence），第 9 章将进一步讨论这一主题。

1.3　前进和提升

我们正处在数据存储选择的寒武纪大爆炸之中，很难准确预测未来会如何发展。但我们可以相当肯定，任何一种特定策略（关系型或其他类型）都不大可能大获全胜。相反，我们将看到越来越多的专用数据库，每种适合一组特定（但肯定有重叠）的理想问题。今天有一些工作需要关系数据库的专业知识（DBA），同样，我们会看到对应的非关系领域的增长。

与编程语言和程序库一样，数据库是另一套工具，每个开发人员都应该知道数据库。每一个好木匠必须了解他的工具箱里有什么。就像所有好的建筑师一样，如果你不熟悉可供你使用的多种选择，就不会成为一名大师。

本书就像是一个车间里的速成课。在本书中，你会挥动锤子，转动电钻，使用射钉枪，并最终能够建立比鸟笼子更大、更复杂的东西。闲话少说，我们来使用我们的第一个数据库：PostgreSQL。

第 2 章

PostgreSQL

PostgreSQL 是数据库世界里的“锤子”。它既广为人知，又容易获得，还很坚固，如果你抡得够猛，它所能解决的问题数量惊人。如果不了解这个最常用的工具，你就不可能成为建筑专家。

PostgreSQL 是一个关系数据库管理系统，即它是以集合理论为基础的系统，在实现上，它定义为一些二维表，表中包含数据行和具有严格数据类型的列。虽然人们对新兴数据库越来越有兴趣，但关系数据库仍然是最流行的数据库，而且这种趋势可能会保持很长一段时间。

关系数据库流行的原因，不仅在于其庞大的特性集（触发器、存储过程、高级索引）、数据的安全性（符合 ACID），或符合大多数人的思维方式（许多程序员以关系的方式说话和思考），还在于它们的查询灵活性。与其他某些数据存储库相比，你不必事先知道要如何使用这些数据。如果关系数据模式是规范的，那么查询就可以很灵活。PostgreSQL 是最好的开源关系数据库例子。

2.1 这就是 Post-greS-Q-L

在本书提到的数据库中，PostgreSQL 是历史最悠久、实战经验最丰富的。它的扩展包括自然语言解析、多维索引、地理查询、自定义数据类型等。它具有高级的事务处理能力，支持十几种不同语言的存储过程，能在各种平台上运行。PostgreSQL 内置支持 Unicode、序列、表继承、子查询，而且是市场上遵循 ANSI SQL 标准最好的关系数据库之一。它快速可靠，可以处理 TB 量级的数据，并且已经在一些高知名度的生产系统上得到验证，如 Skype、法国储蓄银行（CNAF）和美国联邦航空局（FAA）。

> **那么，名字是怎么来的呢**
> 自 1995 年以来，PostgreSQL 就以目前的项目形态存在，但它的起源相当久远。20 世纪 70 年代初，最初的项目产生于加州大学伯克利分校，叫做交互式图形和检索系统（Interactive Graphics and Retrieval System），或简称为“Ingres”。在 20 世纪 80 年代，推出了一个改进版本，post-Ingres，简称为 Postgres。虽然该项目于 1993 年在伯克利大学终结，但开源社区取得了该项目的源码，并将其发布为 PostgreSQL95。后来于 1996 年更名为 PostgreSQL，表示对新的 SQL 标准的支持，此后一直沿用这个名字。

可以用多种方式安装PostgreSQL，这取决于你的操作系统[1]。除了安装核心组件，还需要在PostgreSQL上安装扩展包，用到以下扩展包：`tablefunc`、`dict_xsyn`、`fuzzystrmatch`、`pg_trgm`和`cube`。可以参考网站上的安装指南[2]。

安装 PostgreSQL 之后，使用下面的命令创建一个名为 `book` 的数据库：

```
$ createdb book
```

接下来，我们将在本章中使用 `book` 数据库。运行下面的命令，以确保你需要的扩展包已经正确安装。

```
$ psql book -c "SELECT '1'::cube;"
```

如果你看到一条错误消息，请查看官网的文档，以获得更多的信息。

2.2　第 1 天：关系、CRUD 和联接

我们虽然不会把你当作是一个关系数据库专家，但是确实会假设你曾用过一两个数据库。这些数据库很可能是关系型的。我们将开始创建自己的数据表，并填充数据。然后尝试查询一些行。最后探讨关系数据库中非常重要的表联接。

就像大多数数据库一样，Postgres 提供一个后台服务进程（Backend），它完成所有数据处理工作，还提供一个命令行客户端程序，通过它连接到运行中的服务进程。服务进程默认监听 5432 端口，可以用 `psql` 这个命令行工具连接。

> **数学关系**
> 关系数据库的名称源于它们包含关系（即表），它们是元组（即行）的集合，元组又将属性映射到原子值（例如，`{name: 'Genghis Khan', p.died_at_age: 65}`）。

[1] http://www.postgresQL.org/download/
[2] http://www.postgresQL.org/docs/9.0/static/contrib.html

可用的属性通过头部的属性元组来定义，这些属性映射到某个域或限制的类型（即列；例如，`{name: string, age: int}`）。这是关系结构的要点。

尽管听起来数学味很浓，但是实现比名字所暗示的更具有现实意义。那么，为什么要提到这些？我们正试图说明，关系数据库的关系是因为它的数学基础，不是因为表通过外键彼此“关联”。这样的限制是否存在并不是关键。

虽然许多数学关系你看不到，但模型的力量肯定是蕴藏在数学之中。这种魔法允许用户提出功能强大的查询，然后让系统基于预定义的模式进行优化。RDBMS 基于集合理论的一个分支，名为关系代数，它包括选择（`WHERE...`）、投影（`SELECT...`）、笛卡尔积（`JOIN...`）等操作，如图 2-1 所示。

如果将关系想象为一张物理表（数组的数组，在数据库入门课中无数次重复过），可能在实践中造成痛苦，如编写遍历所有行的代码。关系查询的描述性远胜于此，它源于一个数学分支，名为元组关系演算，可以转换为关系代数。PostgreSQL 和其他的 RDBMS 通过执行这个转换优化了查询，简化了代数运算。你可以看到，图 2-2 中的 SQL，与图 2-1 中的 SQL 是一样的。

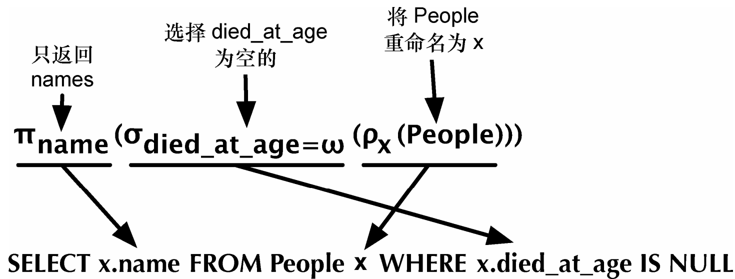


图 2-1　关系代数和 SQL

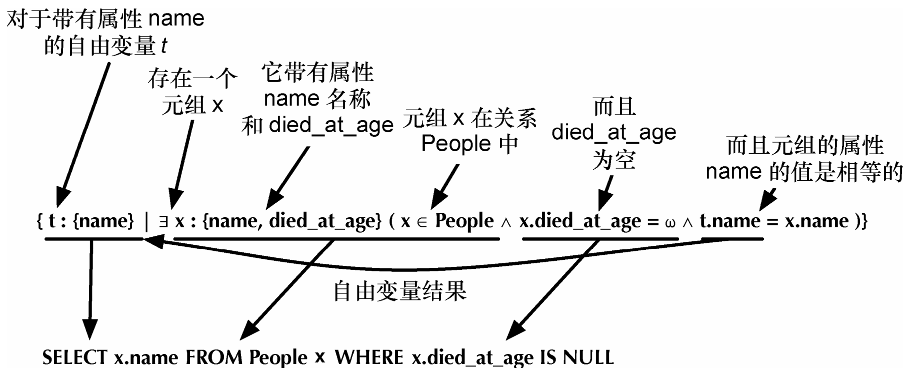


图 2-2　元组关系演算和 SQL

```
$ psql book
```

以管理员用户运行的话，PostgreSQL 的提示符是数据库的名字后面跟一个‘#’，如果是普通用户，后面跟的是‘$’。这个命令行程序的内置文档是所有命令行程序中最好的。输入‘\h’，可以列出有关 SQL 命令的信息，\? 列出以反斜杠开始的 psql 特有命令的帮助信息。可以使用下列方式找到每个 SQL 命令的使用详细信息：

```
book=# \h CREATE INDEX
Command: CREATE INDEX
Description: define a new index
Syntax:
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table [ USING method ]
    ( { column | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | ...
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace ]
    [ WHERE predicate ]
```

在我们深入探讨 PostgreSQL 之前，最好先熟悉这个有用的工具。还可以熟悉一些常见命令值，如 SELECT 或 CREATE TABLE。

2.2.1 从SQL开始

PostgreSQL 遵循 SQL 惯例，称关系为表（TABLE），属性为列（COLUMN），元组为行（ROW）。虽然你可能会遇到一些数学术语，如关系、属性和元组，为了保持一致性，我们将使用这些术语，如关系、属性和元组。有关这些概念的更多信息，请参阅 2.2 节的“数学关系”。

> **关于 CRUD**
>
> CRUD 是一个助记符，帮助记忆数据管理基本操作：创建、读取、更新和删除（Create，Read，Update，Delete）。这些操作一般对应插入新记录（创建），修改现有记录（更新），删除不再需要的记录（删除）。你使用数据库时所有的其他操作（你可以梦想到的任何疯狂查询）都是读操作。如果能进行 CRUD 操作，你就能做任何事。

2.2.2 使用表

PostgreSQL 是关系型的数据管理系统，所以需要事先进行设计。要先设计好数据库的表，然后插入符合数据库定义的数据。

创建表包括为它命名，定义所有列及其类型，以及定义（可选的）约束信息。每张表都应该指定唯一的标识符列，以标识特定的行。该标识符称为主键（PRIMARY KEY）。创

建 countries 表的 SQL 语句如下所示：

```
CREATE TABLE countries (
  country_code char(2) PRIMARY KEY,
  country_name text UNIQUE
);
```

这张新表将存储一些行，其中每一行由两个字节的国家代码作为标识，国家名也是唯一的。这两列都有约束，主键约束 country_code 列不允许有重复的国家代码，所以只有一个 us 和一个 gb 可以存在表中。尽管 country_name 不是主键，但是明确地给予 country_name 类似的唯一性约束。可以用如下语句插入几行来填充这张 counties 表。

```
INSERT INTO countries (country_code, country_name)
VALUES ('us','United States'), ('mx','Mexico'), ('au','Australia'),
       ('gb','United Kingdom'), ('de','Germany'), ('ll','Loompaland');
```

让我们来测试一下唯一性约束。如果尝试添加包含重复的 country_name 的行，就会因为唯一性约束而不允许插入。约束是 PostgreSQL 这样的关系数据库用来确保数据完整的方法。

```
INSERT INTO countries
VALUES ('uk','United Kingdom');
ERROR: duplicate key value violates unique constraint "countries_country_name_key"
DETAIL: Key (country_name)=(United Kingdom) already exists.
```

通过 SELECT...FROM table 语句进行查询，可以验证相关的行是否已经插入。

```
SELECT *
FROM countries;
country_code | country_name
-------------+---------------
us           | United States
mx           | Mexico
au           | Australia
gb           | United Kingdom
de           | Germany
ll           | Loompaland
(6 rows)
```

根据正规的地图，可以知道 Loompaland 不是真实存在的地方，所以让我们从表中删除它。用 WHERE 子句指定要删除的行，country_code 等于 ll 的行将被删除。

```
DELETE FROM countries
WHERE country_code = 'll';
```

只有实际存在的国家留在了 countries 表中，让我们再添加一个 cities 表。为保证所有插入的 country_code 都在 countries 表中，将添加关键字 REFERENCES。因为 country_code 列引用了另一张表的键，所以它称为外键约束。

```
CREATE TABLE cities (
  name text NOT NULL,
  postal_code varchar(9) CHECK (postal_code <> ''),
  country_code char(2) REFERENCES countries,
  PRIMARY KEY (country_code, postal_code)
);
```

这一次，cities 表中的 name 列的约束是不允许其值为 NULL 的。postal_code 列的约束，是其值不能是空字符串（<>表示不等于）。

此外，因为主键唯一地标识一行，所以定义了一个复合键：country_code + postal_code。它们共同作为一行的唯一的标识符。

Postgres 也有丰富的数据类型，刚才看到了三种不同的字符串表示：text（任意长度的字符串），varchar（9）（长度可达 9 个字节的字符串）和 char（2）（正好两个字节的字符串）。

定义了数据表后，让我们插入 Toronto，CA。

```
INSERT INTO cities
VALUES ('Toronto','M4C1B5','ca');
ERROR: insert or update on table "cities" violates foreign key constraint
  "cities_country_code_fkey"
DETAIL: Key (country_code)=(ca) is not present in table "countries".
```

这个操作失败并不是什么坏事！因为 country_code 需要参考 countries，所以 country_ code 必须存在于 countries 表中，这称为保持参照完整性，参见图 2-3，它确保数据始终是正确的。值得指出的是，NULL 对 cities.country_code 是有效的，因为 NULL 代表一个值空缺。如果你不想允许 country_code 引用为 NULL，可以这样定义 cities 表的列：country_code char（2）REFERENCES countries NOT NULL。

现在我们再试试插入一个美国城市的数据。

```
INSERT INTO cities
VALUES ('Portland','87200','us');

INSERT 0 1
```

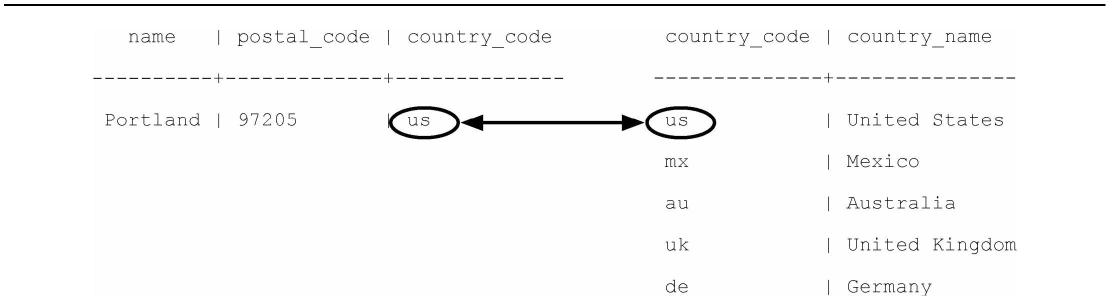


图 2-3　PREFERENCE 关键字约束字段参照另一张表的主键。

当然，这是一次成功的插入。但是我们输入了错误的邮政编码。波特兰（Portland）正确的邮政编码是 97205，但我们不必删除并重新插入，可以直接更新这一行。

```
UPDATE cities
SET postal_code = '97205'
WHERE name = 'Portland';
```

现在已经可以创建、读取、更新、删除表中的行了。

2.2.3　使用联接的查询

在本书中学习的所有其他数据库，也都可以执行 CRUD 操作。但 PostgreSQL 这样的关系数据库有独特的能力，能够在读取表时对表进行联接。联接本质上是以某种方式联接两个独立的表，并返回一张结果表。这有点像拼字游戏，打散单词的字母卡片，重新拼接字母，从而得到新的词。

联接的基本形式是内联接（inner join）。最简单的形式就是，使用 ON 关键字指定匹配的两列（每张表一列）

```
SELECT cities.*, country_name
FROM cities INNER JOIN countries
  ON cities.country_code = countries.country_code;

country_code |   name   | postal_code | country_name
-------------+----------+-------------+---------------
us           | Portland | 97205       | United States
```

联接返回单张表，其中包含 cities 表的所有列的值，再加上匹配的 countries 表中 country_name 的值。

也可以联接诸如 cities 这样有复合主键的表。为了测试复合联接，我创建一张新表，来存储场地（venue）的列表。

某个国家和一个邮政编码组成一个场所。外键必须引用 Cities 表的两个主键列。（MATCH FULL 是一个约束，确保两个值都存在，或两者均为 NULL。）

```
CREATE TABLE venues (
  venue_id SERIAL PRIMARY KEY,
  name varchar(255),
  street_address text,
  type char(7) CHECK ( type in ('public','private') ) DEFAULT 'public',
  postal_code varchar(9),
  country_code char(2),
  FOREIGN KEY (country_code, postal_code)
    REFERENCES cities (country_code, postal_code) MATCH FULL
);
```

其中 venue_id 列是一种常见的主键设置：设置为自动递增整数（1，2，3，4，…）。可以使用 SERIAL 关键字来定义这个标识符（MySQL 有一个类似的构造，称为 AUTO_INCREMENT）。

```
INSERT INTO venues (name, postal_code, country_code)
VALUES ('Crystal Ballroom', '97205', 'us');
```

虽然没有设置 venue_id 的值，但创建行时会填充它。

回到复合联接。联接 enues 表和 cities 表需要用到两个外键列。为了减少输入量，可以在表名后面直接加别名，它们中间的 AS 是可选的（例如，venues v 或 venues AS v）。

```
SELECT v.venue_id, v.name, c.name
FROM venues v INNER JOIN cities c
  ON v.postal_code=c.postal_code AND v.country_code=c.country_code;
venue_id |   name    | name
---------+---------+---------
       1 | Crystal Ballroom | Portland
```

可以选择指定 PostgreSQL 在插入后返回一些列，方法是让请求以 RETURNING 语句结尾。

```
INSERT INTO venues (name, postal_code, country_code)
VALUES ('Voodoo Donuts', '97205', 'us') RETURNING venue_id;
 id
- - - -
  2
```

无须执行另一个查询，就可以得到新插入的 venue_id 值。

2.2.4 外联接

除了内联接，PostgreSQL 也可以执行外联接（outer join）。外联接是合并两张表的一种方式，不论另一张表中是否存在匹配的列值，第一张表的结果总是必须返回。

最简单的方法是举一个例子，但是首先我们需要创建一张名为 `events` 的新表。`events` 表应该有这些列：SERIAL 整数 `event_id`、`title`、`starts` 和 `ends`（类型为时间戳），以及 `venue_id`（引用 `venues` 的外键）。图 2-4 展示了一个数据库的定义图，它涵盖了到目前为止我们创建的所有表。

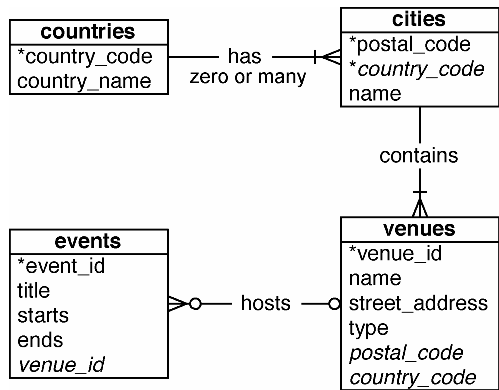


图 2-4　鱼尾纹实体关系图（ERD）

创建 `events` 表后，插入以下值（时间戳作为字符串插入，例如，2012-02-15 17:30），两个节日，以及我们不会详加讨论的一个俱乐部。

```
       title        |       starts        |        ends         | venue_id | event_id
--------------------+---------------------+---------------------+----------+---------
LARP Club           | 2012-02-15 17:30:00 | 2012-02-15 19:30:00 |    2     |        1
April Fools Day     | 2012-04-01 00:00:00 | 2012-04-01 23:59:00 |          |        2
Christmas Day       | 2012-12-25 00:00:00 | 2012-12-25 23:59:00 |          |        3
```

我们先来做一个查询，使用内联接返回一个事件的标题和场地名称（INNER JOIN 中的 INNER 并不是必需的，所以这里省略它）。

```
SELECT e.title, v.name
FROM events e JOIN venues v
  ON e.venue_id = v.venue_id;
    title     |     name
--------------+--------------
LARP Club     | Voodoo Donuts
```

只有列值匹配，INNER JOIN 才会返回一行。因为不能有空的 venues.venue_id，所以两个空 events.venue_id 没有关联到任何事情。要查询所有的事件，不管它们是否有场地，我们需要一个左外连接（LEFT OUTER JOIN，简写为 LEFT JOIN）。

```
SELECT e.title, v.name
FROM events e LEFT JOIN venues v
ON e.venue_id = v.venue_id;
       title      |      name
------------------+---------------
 LARP Club        | Voodoo Donuts
 April Fools Day  |
 Christmas Day    |
```

如果你需要反过来，返回所有的场地和匹配的事件，就要用 RIGHT JOIN。最后，还有 FULL JOIN，这是 LEFT 和 RIGHT 的联合；保证能得到每张表中的所有值，列匹配时就会联接。

2.2.5　使用索引快速查找

PostgreSQL 的速度（和任何其他 RDBMS 一样）源于其高效的数据块管理、尽可能少的磁盘块读取、查询优化等技术。如果从 events 表选择 title 为 Christmas Day 的行，则需要进行全表扫描，以返回相关的结果。如果没有索引，就必须从磁盘读取每一行，才能知道是否是匹配行。参见图 2-5。

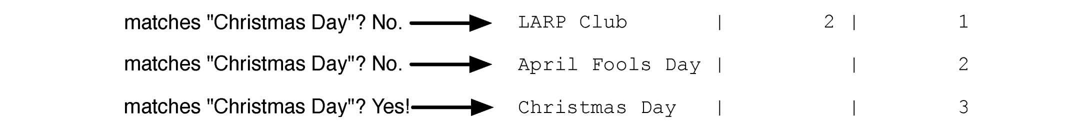


图 2-5　全表扫描是找到匹配的数据最慢的方法

索引是一个特殊的数据结构，目的是避免执行查询时进行全表扫描。当运行 CREATE TABLE 命令时，你可能注意到这样一条消息：

```
CREATE TABLE / PRIMARY KEY will create implicit index "events_pkey" \
for table "events"
```

PostgreSQL 自动在主键上创建索引，以主键的列值为索引的键，索引的值则指向磁盘上的一行，如图 2-6 所示。采用 UNIQUE 关键字，是强制在表中一列上创建索引的另一种方式。

可以使用 CREATE INDEX 命令明确地添加一个哈希索引，其中每个值必须是唯一的（就像一个哈希或映射）。

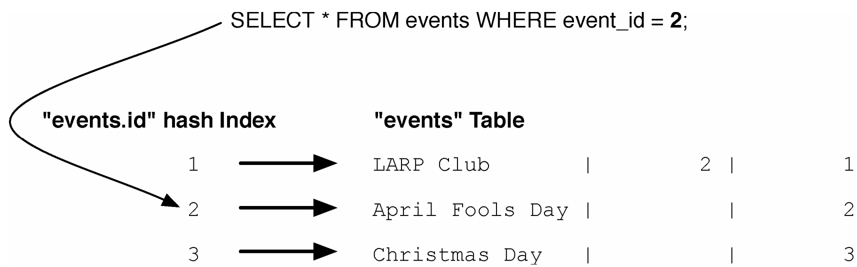


图 2-6　利用索引的查询指向精确的行而无需进行表扫描

```
CREATE INDEX events_title
  ON events USING hash (title);
```

对于操作符为小于/大于/等于这样的匹配查询，我们希望索引比简单的哈希更灵活，如 B 树索引（见图 2-7）。考虑用一个查询来查找 4 月 1 日或之后发生的所有事件。

```
SELECT *
FROM events
WHERE starts >= '2012-04-01';
```

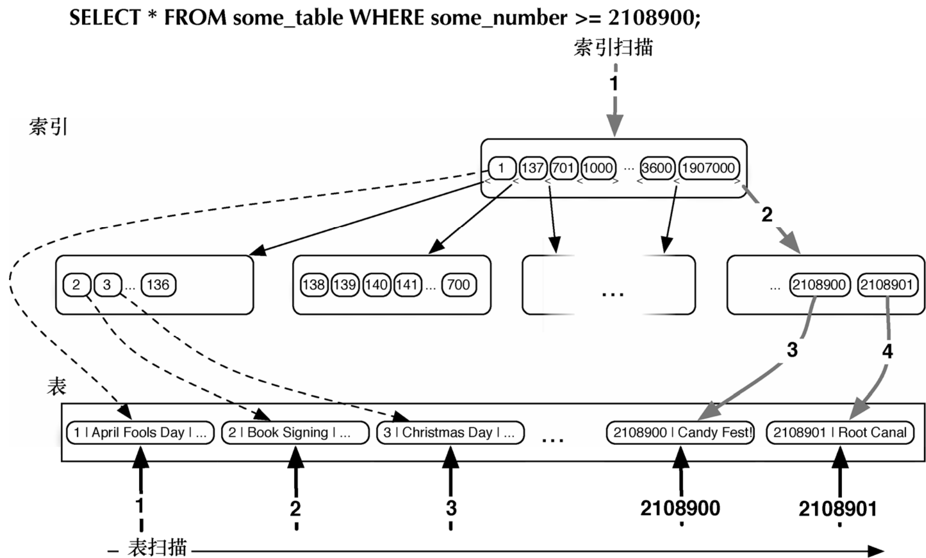


图 2-7　B 树索引可以匹配范围查询

对于这样的操作，树是一个完美的数据结构。要对 starts 列创建 B 树索引，使用下面的命令：

```
CREATE INDEX events_starts
  ON events USING btree (starts);
```

这样对日期范围的查询将可以避免全表扫描。当扫描数百万或数十亿行时，上述查询的性能差异会很大。

可以用下面的命令，列出数据模式中的所有索引：

```
book=# \di
```

值得注意的是，当对列创建一个 FOREIGN KEY 约束时，PostgreSQL 将自动在目标列创建索引。即使你不喜欢使用数据库约束，也会经常发现自己需要在进行联接的列上创建索引，以便加快基于外键的表联接。

2.2.6 第 1 天总结

我们今天快速介绍了许多内容，涉及很多方面。总结如下：

术 语	定 义
列	具有某种类型的值的域，有时也称为属性
行	一组列值组成的一个对象，有时也称为一个元组
表	一组具有相同的列的行，有时也称为一个关系
主键	指向一个特定的行的唯一的值
CRUD	创建、读取、更新、删除
SQL	结构化查询语言，关系数据库的标准语言
联接	按一些匹配的列结合两张表
左联接	按一些匹配的列结合两张表，如果没有匹配左表的内容，就以 NULL 补足
索引	优化选择特定的一组列的数据结构
B-树	一个很好的标准索引；值存储为一个平衡树数据结构；非常灵活

四十多年来，关系数据库已经成为事实上的数据管理策略，我们中的很多人在其发展的中途，开始了自己的职业生涯。因此，我们通过一些基本的 SQL 查询，初步探讨了关系模型的一些核心概念。明天我们将详细说明这些基本概念。

2.2.7 第 1 天作业

查找

1．将 PostgreSQL 官网的常见问题集（FAQ）和官方文档保存为书签。

2．熟悉 PSQL 的命令行`\?`和`\h`的输出信息。

3．在 FOREIGN KEY 的定义中文档中找到 MATCH FULL 是什么意思。

完成

1．从 pg_class 中查询我们创建的所有表（仅我们创建的）。

2．编写一个查询，找到 LARP Club 事件对应的国家名。

3．修改 venues 表，增加一个名为 active 的列，该列为布尔类型，默认值是 TRUE。

2.3　第 2 天：高级查询、代码和规则

昨天，我们看到了如何定义数据库表，然后插入数据，更新和删除行，以及读数据。今天我们将更深入探讨 PostgreSQL 查询数据的各种方法。

我们将看到如何对相似的值归类，在服务器端执行代码，并使用视图（view）和规则（rule）创建自定义接口。在这一天的最后，我们将利用 PostgreSQL 的一个扩展包翻转表头。

2.3.1　聚合函数

聚合查询按照一些共同的标准将多行的结果分组。它可以简单到统计一张表的行数，或计算某些数值列的平均值。它们是强大的 SQL 工具，也很有趣。

我们来尝试一些聚合函数，但在此之前我们需要在数据库中有更多的数据。在 countries 表中加入你自己的国家，在 cities 表中加入你自己的城市，并以自己的地址作为场地（这里称为 My Place），然后添加一些记录到 events 表中。

下面是一个简单的 SQL 技巧：在子查询里通过更加可读的 title 来获得 venue_id，而不用直接给出 venue_id。如果 Moby（白鲸）正在 Crystal Ballroom 上演，可以这样设置 venue_id：

```
INSERT INTO events (title, starts, ends, venue_id)
  VALUES ('Moby', '2012-02-06 21:00', '2012-02-06 23:00', (
    SELECT venue_id
    FROM venues
    WHERE name = 'Crystal Ballroom'
  )
);
```

用以下数据填入 events 表（要在 PostgreSQL 里输入 Valentine's Day，可以用双撇号转义，如 Heaven''s Gate）。

```
       title      |       starts        |        ends         |      venue
------------------+---------------------+---------------------+-----------------
 Wedding          | 2012-02-26 21:00:00 | 2012-02-26 23:00:00 | Voodoo Donuts
 Dinner with Mom  | 2012-02-26 18:00:00 | 2012-02-26 20:30:00 | My Place
 Valentine's Day  | 2012-02-14 00:00:00 | 2012-02-14 23:59:00 |
```

设置好数据后，我们尝试一些聚合查询。最简单的聚合函数是 count()，它的意思不言自明。下面统计所有包含 Day 关键字的标题（注：%是 LIKE 搜索中的通配符），可以得到结果为 3。

```
SELECT count(title)
FROM events
WHERE title LIKE '%Day%';
```

要在 Crystal Ballroom 发生的所有事件中，获得最早的开始时间和最晚的结束时间，就要使用 min()（返回最小值）和 max()（返回最大值）。

```
SELECT min(starts), max(ends)
FROM events INNER JOIN venues
  ON events.venue_id = venues.venue_id
WHERE venues.name = 'Crystal Ballroom';
          min          |        max
----------------------+-------------------
 2012-02-06 21:00:00  | 2012-02-06 23:00:00
```

虽然聚合函数很有用，但只用它们是不够的。如果我们想统计每个场地的所有事件，就可以为每个场地 ID 写出如下语句：

```
SELECT count(*)FROM events WHERE venue_id = 1;
SELECT count(*)FROM events WHERE venue_id = 2;
SELECT count(*)FROM events WHERE venue_id = 3;
SELECT count(*)FROM events WHERE venue_id = IS NULL;
```

随着场地的数量增长，这种写法是很让人生厌的（甚至行不通）。因此需要用到 GROUP BY 命令。

2.3.2 分组

通过 GROUP BY 可以简单地完成前面查询。使用 GROUP BY 的时候，你告诉 Postgres 对行进行归类，然后对这些组执行一些聚合函数（如 count()）。

```
SELECT venue_id, count(*)
FROM events
```

```
GROUP BY venue_id;

 venue_id| count
---------+------
        1|     1
        2|     2
        3|     1
         |     3
```

结果看起来不错，但是能否用 count() 函数作为结果的过滤条件？当然可以。GROUP BY 项有其自己的过滤关键字：HAVING。HAVING 和 WHERE 子句类似，只不过它可以用聚合函数作为过滤条件（而 WHERE 不能）。下面的查询找出最热门的场地，它们有两个或更多事件：

```
SELECT venue_id,count(*)
FROM events
GROUP BY venue_id
HAVING count(*) >= 2 AND venue_id IS NOT NULL;

 venue_id| count
---------+------
       2 |     2
```

也可以不带任何聚合函数使用 GROUP BY。如果在一列上用 SELECT...FROM... GROUP BY 查询，就会得到所有不重复的值。

```
SELECT venue_id FROM events GROUP BY venue_id;
```

这种分组归类非常常见，所以 SQL 有一个更为简单的命令，即 DISTINCT 关键词。

```
SELECT DISTINCT venue_id FROM events;
```

这两个查询的结果是相同的。

2.3.3　窗口函数

如果你曾经在生产系统中使用过关系数据库，你应该会熟悉聚合查询。聚合查询是一种常见的 SQL 元素，但窗口函数却不那么常见（PostgreSQL 是少数几个实现了窗口函数的开源数据库之一。）

窗口函数与 GROUP BY 查询是类似的，它们允许你对多行执行聚合函数。所不同的是，窗口函数允许使用内置的聚合函数，而不要求将每个字段分组成单行。

如果试图不对 title 列分组，却要在结果显示这个字段，将会报错。

```
SELECT title, venue_id, count(*)
```

```
FROM events
GROUP BY venue_id;

ERROR: column "events.title" must appear in the GROUP BY clause or \
       be used in an aggregate function
```

要按 venue_id 对行计数，但是对于一个 venue_id 可能有两个不同的活动 LARP Club 和 Wedding Postgres 不知道要显示哪个活动名。

虽然 GROUP BY 子句为每个匹配的组返回一个记录，而窗口函数却可以为每行返回一个不同的记录，图 2-8 描述了这种关系。让我们来看一个例子，这正是窗口函数所能有效解决的场景。

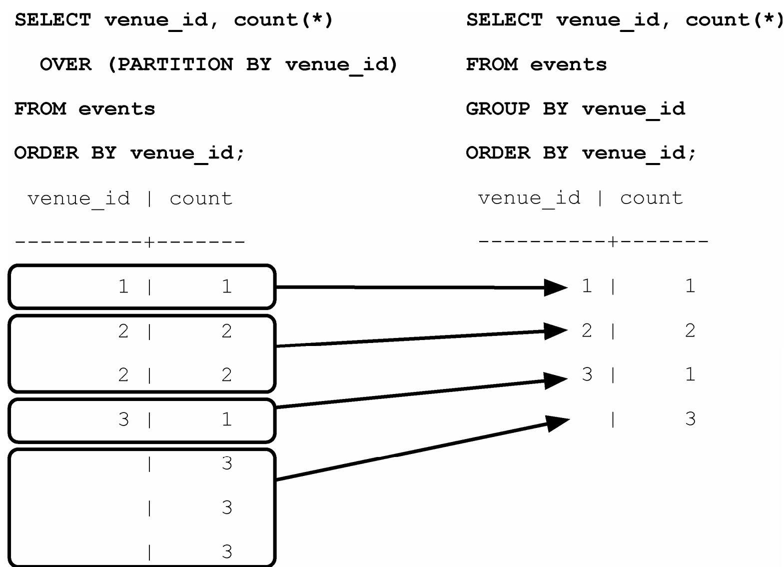


图 2-8　窗口函数的结果不折叠每个组的结果

窗口函数返回所有匹配的记录，并复制所有聚合函数的结果：

```
SELECT title, count(*) OVER (PARTITION BY venue_id) FROM events;
```

我们倾向于认为 PARTITION BY 和 GROUP BY 类似，但它不会在 SELECT 属性列表之后，再对结果分组（从而将结果合并成较少的行），而是像其他所有字段一样返回分组的值（根据分组变量进行计算，但其他方面就是另一个属性）。或者按 SQL 的表达方式，它超越（OVER）结果集的分区（PARTITION），返回聚合函数的结果。

2.3.4　事务

事务保障了关系数据库的一致性。事务的准则是，要么全部成功，要么全部失败。事

务确保一组命令中的每一条命令都执行。如果过程中间发生了任何失败，所有的命令将回滚，就像它们从未发生过一样。

PostgreSQL 的事务遵循 ACID，它代表原子性（Atomic，所有的操作都成功或都没有做），一致性（Consistent，数据将始终处于完整的状态，没有不一致的状态），隔离性（Isolated，事务互相之间不干扰），以及持久性（Durable，即使在服务器崩溃以后，提交的事务都是安全的）。我们应该注意，ACID 中的一致性不同于 CAP 中的一致性（附录 B 介绍了 CAP 理论）。

可以将任何事务的命令置于 BEGIN TRANSACTION 块内。为了验证原子性，将使用 ROLLBACK 命令终止事务。

```
BEGIN TRANSACTION;
  DELETE FROM events;
ROLLBACK;
SELECT * FROM events;
```

event 表里的所有的活动依然存在。如果要修改两个表，并希望它们保持同步，事务就很有用。最典型的例子是一个银行借记/贷记系统，其中钱从一个账户转移到另一个账户：

```
BEGIN TRANSACTION;
   UPDATE account SET total=total+5000.0 WHERE account_id=1337;
   UPDATE account SET total=total-5000.0 WHERE account_id=45887;
END;
```

在 MySQL 中的 GROUP BY

在 MySQL 中，如果你试图 SELECT 一些没有在 GROUP BY 中限定的列，你可能会吃惊地看到，它有结果。这让我们开始怀疑窗口函数的必要性。但更细致地检查 MySQL 返回的数据之后，我们发现它返回的只是一些随机的数据行和计数，并非所有相关的结果。一般来说，这是没有用的（并且可能相当危险）。

不可避免的事务

到现在为止，在 psql 中执行的每条命令都隐式地包裹在事务中。如果你执行一条命令，如 DELETE FROM account WHERE total < 20;，并且数据库在删除的中途崩溃，你不会被迫接受半张表。当你重新启动数据库服务器时，该命令将回滚。

如果在两次更新之间发生意外，这家银行就会损失 5000 美元。但是，如果操作放在一个事务块中，即使服务器爆炸了，最初的更新也会被回滚。

厂商锁定是什么

在关系数据库的全盛时期，厂商锁定是技术方面的瑞士军刀。可以在数据库中存储几乎任何内容，甚至用它们对整个项目编程（例如，Microsoft Access）。少数提供这个软件的公司鼓励用户使用它们专有的差异，然后利用这种公司依赖性，收取巨额的许可证和咨询费。这就是可怕的厂商锁定，在 20 世纪 90 年代和 21 世纪初，新的编程方法试图缓解这种情况。

然而，在他们热衷于保持厂商中立时，产生了一些准则，如“数据库中不含逻辑”。这是一种耻辱，因为关系数据库能够胜任多种不同的数据管理方式。厂商锁定并没有消失。本书探讨的许多动作与具体实现高度相关。但是，先要知道如何充分使用数据库，然后再决定是否跳过存储过程这样的工具。

2.3.5　存储过程

到现在为止，我们看到的每条命令都是声明性的，但有时我们需要运行一些代码。这时你必须做出决定：在客户端执行代码，还是在数据库端执行代码。

存储过程可以通过巨大的架构代价来取得巨大的性能优势。使用存储过程可以避免将数千行数据发送到客户端应用程序，但也让应用程序代码与该数据库绑定，因此，不应该轻易决定使用存储过程。

先把上面的警告放在一边，来创建一个过程（或 FUNCTION），它简化了向 event 表插入记录的工作，无需 venue_id，就可以插入在某个场地举行的活动。如果场地不存在，会先创建它，并在新的事件中引用它。此外，为了用户友好，函数会返回一个布尔值，表明添加新场地是否成功。

postgres/add_event.sql

```
CREATE OR REPLACE FUNCTION add_event(i_title text, i_starts timestamp,
  i_ends timestamp, venue text, postal varchar(9), country char(2) )
RETURNS boolean AS $$
DECLARE
  did_insert boolean := false;
  found_count integer;
  the_venue_id integer;
BEGIN
  SELECT venue_id INTO the_venue_id
  FROM venues v
  WHERE v.postal_code=postal AND v.country_code=country AND v.name ILIKE venue
  LIMIT 1;

  IF the_venue_id IS NULL THEN
```

选择执行数据库代码

本书将多次探讨这个主题，这是第一次：代码属于应用程序还是属于数据库？这是一个困难的决定，对每个应用程序，你都会有不同的答案。

好处是性能常常会提高一个数量级。例如，你可能有一个复杂的、应用程序相关的计算，要求自定义代码。如果计算涉及许多行数据，存储过程让你不必传输数千行数据，只要传一个结果。这样做的代价是割裂应用程序，你的代码和测试将跨越两种不同的编程范式（客户端和服务器）。

```
    INSERT INTO venues (name, postal_code, country_code)
    VALUES (venue, postal, country)
    RETURNING venue_id INTO the_venue_id;

    did_insert := true;
  END IF;

  -- Note: not an"error",as in some programming languages
  RAISE NOTICE 'Venue found %', the_venue_id;

    INSERT INTO events (i_title, i_starts, i_ends, i_venue_id)
    VALUES (title, starts, ends, the_venue_id);

    RETURN did_insert;
  END;
  $$ LANGUAGE plpgsql;
```

如果你不喜欢用键盘输入所有的代码，通过以下命令行参数，可以将这个外部文件导入当前数据库。

```
book=# \i add_event.sql
```

因为这是第一次使用场地 Run’s House，下面的操作应该返回 t（成功）。它只有一次往返，这避免了客户端 SQL 命令到数据库的两次往返（一次查询，然后是一次插入）。

```
SELECT add_event('House Party', '2012-05-03 23:00',
  '2012-05-04 02:00', 'Run''s House', '97205', 'us');
```

在我们所写的存储过程中使用的语言是PL/pgSQL（即Procedural Language/PostgreSQL）。全面介绍其细节超出了本书的范围，但是你可以在PostgreSQL的官方在线文档[1]中看到更多有关内容。

除了 PL/pgSQL，PostgreSQL 还支持三种更核心的语言编写程序：Tcl、Perl 和 Python。社区还开发了 Ruby、Java、PHP、Scheme，以及官方文档列出的十多种语言的扩展模块。

[1] http://www.PostgreSQL.org/docs/9.0/static/plpgsql.html

试试这个 shell 命令：

```
$ createlang book --list
```

它会列出在你的数据库中安装的语言，createlang命令也可以用来添加新的语言。可以在线找到该命令。[1]

2.3.6　触发器

当插入或更新这样的事件发生时，触发器会自动调用存储过程。它们允许数据库在数据变化的时候，强制执行一些必要的操作。

下面创建一个新的 PL/pgSQL 函数，当活动信息 event 更新的时候，都会记录相应的日志（我们想要确保没有人改变事件之后又不承认）。首先，创建一个 logs 表以记录活动信息的变化，这里没有必要使用主键，因为这只是日志。

```
CREATE TABLE logs (
  event_id integer,
  old_title varchar(255),
  old_starts timestamp,
  old_ends timestamp,
  logged_at timestamp DEFAULT current_timestamp
);
```

接下来，创建一个函数，将更新前的数据写入日志。OLD 变量代表更新前的行（我们将很快就会看到 NEW 代表新输入的行的值）。在返回之前，在屏幕上输出一条带 event_id 的信息。

```
postgres/log_event.sql
CREATE OR REPLACE FUNCTION log_event() RETURNS trigger AS $$
DECLARE
BEGIN
  INSERT INTO logs (event_id, old_title, old_starts, old_ends)
  VALUES (OLD.event_id, OLD.title, OLD.starts, OLD.ends);

  RAISE NOTICE 'Someone just changed event #%', OLD.event_id;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

[1] http://www.PostgreSQL.org/docs/9.0/static/app-createlang.html

最后，创建触发器，可以在任意行更新后记录相应变更。

```
CREATE TRIGGER log_events
  AFTER UPDATE ON events
  FOR EACH ROW EXECUTE PROCEDURE log_event();
```

现在，我们在 Run’s House 的聚会将比计划的提前结束。下面更新这个事件。

```
UPDATE events
SET ends='2012-05-04 01:00:00'
WHERE title='House Party';

NOTICE: Someone just changed event #9
```

而且原来的结束时间记入了日志。

```
SELECT event_id, old_title, old_ends, logged_at
FROM logs;

event_id |  old_title  |      old_ends       |       logged_at
---------+------------+---------------------+-----------------------
      9  | House Party| 2012-05-04 02:00:00 | 2011-02-26 15:50:31.939
```

触发器还可以在更新之前以及插入之前或之后创建。[1]

2.3.7 视图

如果复杂查询的结果用起来就像其他任何表一样，那岂不是太棒了？这就是 VIEW 的用途。与存储过程不同，它们不是执行的函数，而是查询的别名。

在我们的数据库中，所有节日都包含单词 Day，并且没有场地信息。

```
postgres/holiday_view_1.sql
CREATE VIEW holidays AS
  SELECT event_id AS holiday_id, title AS name, starts AS date
  FROM events
  WHERE title LIKE '%Day%' AND venue_id IS NULL;
```

可以看到，创建视图很简单，只要在查询前面加上“CREATE VIEW some_view_name AS”。

现在，可以像查询任何其他表一样查询 holidays，其后面是普通不过的 events 表。作为证明，向 events 添加 2012-02-14 的情人节（Valentine’s Day），并查询节日视图 holidays。

[1] http://www.PostgreSQL.org/docs/9.0/static/triggers.htmla

```
SELECT name, to_char(date, 'Month DD, YYYY') AS date
FROM holidays
WHERE date <= '2012-04-01';
       name       |       date
------------------+-------------------
 April Fools Day  | April    01, 2012
 Valentine's Day  | February  14,2012
```

视图是强大的工具，它可以以一种简单的方式访问复杂查询的数据。下面的查询可能非常复杂，但是你看到的全部只是一张表。

如果你想在视图中添加一个新列，那么，你只能修改底层的表。修改 events 表，使它有一组相关的颜色。

```
ALTER TABLE events
ADD colors text ARRAY;
```

因为 holidays 有与它们相关联的颜色组，那我们就修改视图的查询，以包含 colors 数组。

```
CREATE OR REPLACE VIEW holidays AS
  SELECT event_id AS holiday_id, title AS name, starts AS date, colors
  FROM events
  WHERE title LIKE '%Day%' AND venue_id IS NULL;
```

现在要为选定的节日设置一个颜色字符串数组，但遗憾的是，我们不能直接更新视图。

```
UPDATE holidays SET colors = '{"red", "green"}' where name = 'Christmas Day';

ERROR: cannot update a view
HINT: You need an unconditional ON UPDATE DO INSTEAD rule.
```

看起来需要一条规则。

2.3.8　规则是什么

规则是对如何修改解析过的查询树的描述。Postgres 每次运行一条 SQL 语句，它将语句解析成查询树（一般称为抽象语法树）。

树的枝和叶是运算符和值，在执行前，树会被遍历、删减，并以其他方式修改。这棵树可以被 Postgres 规则重写，然后发送到查询规划器（它也以某种方式重写这棵树，以达到优化性能运行效果），最后会把最终的命令发送到执行器。参见图 2-9。

需要找出的是，诸如 holidays 这样的视图其实就是一条规则。

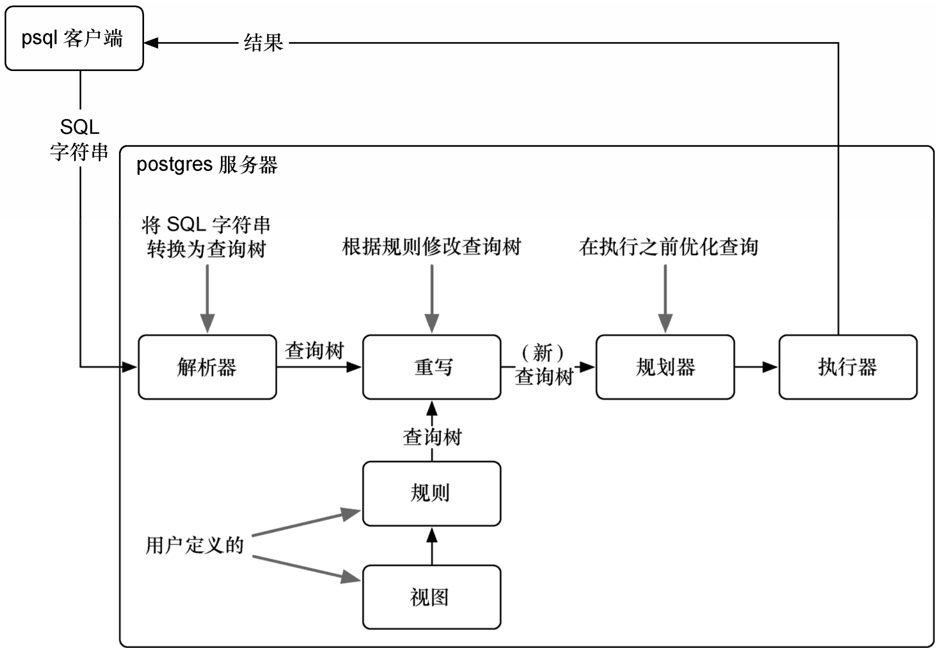


图 2-9　在 PostgreSQL 中 SQL 如何执行

用 EXPLAIN 命令看一看 holidays 视图的执行计划，我们可以证明这一点（注意，Filter 是 WHERE 子句，Output 为列的列表）。

```
EXPLAIN VERBOSE
   SELECT *
   FROM holidays;

                                  QUERY PLAN
-------------------------------------------------------------------------------
Seq Scan on public.events (cost=0.00..1.04 rows=1 width=57)
  Output: events.event_id, events.title, events.starts, events.colors
  Filter: ((events.venue_id IS NULL)AND ((events.title)::text~~ '%Day%'::text))
```

如果对定义 holidays 视图的查询语句执行 EXPLAIN VERBOSE，并和上述结果比较，我们会发现它们在功能上是等价的。

```
EXPLAIN VERBOSE
  SELECT event_id AS holiday_id,
    title AS name, starts AS date, colors
  FROM events
  WHERE title LIKE '%Day%' AND venue_id IS NULL;
                                           QUERY PLAN
-------------------------------------------------------------------------------
Seq Scan on public.events (cost=0.00..1.04 rows=1 width=57)
```

```
Output: event_id, title, starts, colors
Filter: ((events.venue_id IS NULL)AND ((events.title)::text~~ '%Day%'::text))
```

所以，为了允许更新 holidays 视图，需要定义一条规则，告诉 PostgreSQL 在 UPDATE 的时候做什么操作。规则将捕捉对 holidays 视图的更新，从 NEW 与 OLD 的伪关系取值，并在 events 上执行更新。NEW 看作是包含即将更新的值的表，而 OLD 则是包含查询的值。

```
postgres/create_rule.sql
CREATE RULE update_holidays AS ON UPDATE TO holidays DO INSTEAD
  UPDATE events
  SET title = NEW.name,
      starts = NEW.date,
      colors = NEW.colors
  WHERE title = OLD.name;
```

有了这条规则，现在可以直接更新 holidays。

```
UPDATE holidays SET colors = '{"red","green"}' where name = 'Christmas Day';
```

接下来将 2013-01-01 的 New Years Day 插入 holidays 中。正如我们所料，这也需要一条规则。没有问题。

```
CREATE RULE insert_holidays AS ON INSERT TO holidays DO INSTEAD
  INSERT INTO ...
```

我们即将讨论其他内容，但是如果你想多练习 RULE 的用法，可以尝试添加 DELETE RULE。

2.3.9 联表分析

作为今天最后的练习，下面将要建立一个事件月历，对一年中各月发生的事件计数。这种操作通常由一个数据透视表（pivot table）完成。这些构造以另外某种输出为“中心”，对数据分组。在例子中，中心是月份列表。我们将使用 crosstab() 函数创建数据透视表。

首先设计一个查询，来统计每年中每月里的事件数量。PostgreSQL 提供了 extract() 函数，它返回日期或时间戳的某些部分，我们利用它来对事件进行归类。

```
SELECT extract(year from starts) as year,
  extract(month from starts) as month, count(*)
FROM events
GROUP BY year, month;
```

为了使用 crosstab()，查询必须返回三列：rowid、category 和 value。我们将把 year 作为一个 ID，这意味着其他域是类别（月）和值（计数）。

crosstab()函数需要另一组值代表月。根据这组值，该函数知道需要多少列。这组值将成为列（透视所依据的表）。现在，我们创建一张表，存储临时的数字列表。

```
CREATE TEMPORARY TABLE month_count(month INT);
INSERT INTO month_count VALUES (1),(2),(3),(4),(5),(6),(7),(8),(9),(10),(11),(12);
```

现在，我们准备好以两个查询来调用 crosstab()。

```
SELECT * FROM crosstab(
  'SELECT extract(year from starts) as year,
  extract(month from starts) as month, count(*)
  FROM events
  GROUP BY year, month',
  'SELECT * FROM month_count'
);

ERROR: a column definition list is required for functions returning "record"
```

糟糕，发生错误了。

它可能看起来神秘，其实它是在说，该函数返回一组记录（行），但不知道如何标记它们。事实上，它甚至不知道它们是什么样的数据类型。

请记住，数据透视表使用月份作为类别，但这些月份只是整数。所以，这样定义它们：

```
SELECT * FROM crosstab(
  'SELECT extract(year from starts) as year,
  extract(month from starts) as month, count(*)
  FROM events
  GROUP BY year, month',
  'SELECT * FROM month_count'
) AS (
  year int,
  jan int, feb int, mar int, apr int, may int, jun int,
  jul int, aug int, sep int, oct int, nov int, dec int
) ORDER BY YEAR;
```

有一个 year 列（这是行 ID）和 12 个代表月份的列。

```
year | jan | feb | mar | apr | may | jun | jul | aug | sep | oct | nov | dec
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----
2012 |     |  5  |     |  1  |  1  |     |     |     |     |     |     |  1  |
```

```
 2013 |      |  1  |     |     |     |     |     |     |
```

为了要看到下一年的事件计数，在另一年里增加更多的事件。再次运行 `crosstab()` 函数，再看看你的大作。

2.3.10 第 2 天总结

今天完成了 PostgreSQL 的基础内容。我们开始看到，PostgreSQL 不仅仅是一个存储和查询简单数据类型的服务器；它还是一个数据管理引擎，可以重新格式化输出数据、存储各种数据类型（如数组）、执行逻辑，并提供足够的能力来重写传入的查询。

第 2 天作业

求索

1. 在 PostgreSQL 文档中找到聚合函数列表。
2. 找到一个与 PostgreSQL 进行交互的 GUI 程序，例如 Navicat。

实践

1. 创建一条规则，把对场地的删除，改为将 active 标志（在第 1 天作业中创建的）设置为 FALSE。
2. 临时表不是实现事件月历数据透视表的最好方式。`generate_series(a,b)` 函数返回一组，从 `a` 到 `b` 的记录。用它来替换 `month_count` 表的 SELECT。
3. 建立一个数据透视表，显示在单个月份中的每一天，像一个标准的单月日历那样，其中这个月的每个星期是一行，每一天的名字作为表头（七天，从周日开始，到周六结束）。每天应包含该日期的事件数量，如果没有事件发生，该天应保持空白。

2.4 第 3 天：全文检索和多维查询

第 3 天我们将要学习如何使用多个工具，建立一个电影查询系统。首先，使用 PostgreSQL 自带的多种模糊字串匹配功能，搜索演员/电影的名字。然后，创建一个根据我们喜欢电影的风格进行推荐的系统，以学习 cube 扩展模块。所有这些都是 PostgreSQL 特有的扩展模块，不是 SQL 标准定义的内容。

在设计关系数据库的数据模式时，通常会从实体关系图开始。对将要创建的管理电影、电影风格和演员的个人电影推荐系统，建立如图 2-10 的实体模型。

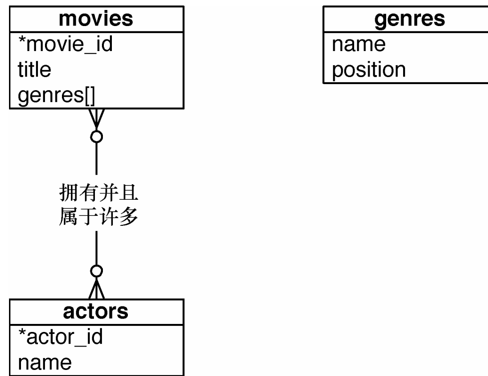


图 2-10 电影推荐系统

提醒一下，在第 1 天，我们安装了几个扩展模块。今天我们会全部用到。我们需要用到的扩展模块列表如下：`tablefunc`、`dict_xsyn`、`fuzzystrmatch`、`pg_trgm` 和 `cube`。

首先创建数据库。一般来说，最好在外键上创建索引以加快反向查找（如通过演员信息，反向查找到其参演的所有电影）。除此之外也需要在 `movies_actors` 这样的联接表上设置一个 UNIQUE 约束，以避免重复的联接值。

postgres/create_movies.sql
```
CREATE TABLE genres (
        name text UNIQUE,
        position integer
);
CREATE TABLE movies (
        movie_id SERIAL PRIMARY KEY,
        title text,
        genre cube
);
CREATE TABLE actors (
        actor_id SERIAL PRIMARY KEY,
        name text
);
CREATE TABLE movies_actors (
        movie_id integer REFERENCES movies NOT NULL,
        actor_id integer REFERENCES actors NOT NULL,
        UNIQUE (movie_id, actor_id)
);
CREATE INDEX movies_actors_movie_id ON movies_actors (movie_id);
```

```
CREATE INDEX movies_actors_actor_id ON movies_actors (actor_id);
CREATE INDEX movies_genres_cube ON movies USING gist (genre);
```

你也可以下载本书配套的 movies_data.sql 脚本，并将其导入到数据库来创建相关的表。关于 genre cube 的任何问题，将在本节稍后讨论。

2.4.1 模糊搜索

系统能够支持文本检索功能，就意味系统要面对不确切的输入。你必须能够接受“Brid of Frankstein”这样的错别字。有时候，用户可能不记得“J. Roberts”的全名。在另一些时候，我们只是不知道如何拼写“Benn Aflek”。我们会看到通过一些PostgreSQL扩展包，可以使全文检索变得方便。值得注意的是，随着讨论的深入，这种字符串匹配的功能模糊了关系查询和Lucene[1]这样的搜索框架之间的差异。虽然有些人可能会觉得全文搜索这样的功能属于应用程序代码，但将这些扩展模块放到存放数据的数据库中，可以带来性能和管理上的好处。

2.4.2 SQL标准的字符串匹配

PostgreSQL 有很多方法进行文本匹配，但两大默认方法是 LIKE 和正则表达式。

1. LIKE 和 ILIKE

LIKE 和 ILIKE（不区分大小写的 LIKE）是文本搜索最简单的形式。它们在关系数据库中是相当普遍的。LIKE 比较列值和给定的模式字符串。%和_字符是通配符。%表示匹配任意数量的任何字符，而_表示只匹配一个字符。

```
SELECT title FROM movies WHERE title ILIKE 'stardust%';
        title
--------------------
 Stardust
 Stardust Memories
```

有个小技巧，如果想确保子串 stardust 不在字符串的末尾，可以使用下划线（_）字符。

```
SELECT title FROM movies WHERE title ILIKE 'stardust_%';
        title
--------------------
Stardust
Stardust Memories
```

[1] http://lucene.apache.org/

虽然在简单的场景下，这个功能非常有用，但是 LIKE 只能用于简单的通配符。

2. 正则表达式

更强大的字符串匹配语法是正则表达式（regex）。因为许多数据库都支持它们正则表达式在本书中常常出现。正则表达式涉及的内容很广且复杂甚至还有一些书专门介绍如何编写强大的表达式。所以本书很难对这方面内容深入讨论。PostgreSQL（基本上）采用 POSIX 风格的正则表达式。

在 Postgres 里，正则表达式的匹配字串由～运算符开始，还可以带有可选的!（意思是不匹配）和*（意思是不区分大小写）。因此，要统计所有不以 the 开始的电影（不区分大小写），可以用下面的查询。匹配字符串内的字符是正则表达式。.

```
SELECT COUNT(*)FROM movies WHERE title !～* '^the.*';
```

可以为匹配前面查询的模式创建字符串索引，方法是创建一个 text_pattern_ops 运算符类索引，只要值以小写形式索引。

```
CREATE INDEX movies_title_pattern ON movies (lower(title) text_pattern_ops);
```

因为标题是文本类型，所以使用了 text_pattern_ops 类型的索引。如果需要对 varchar、char 或 name 类型的字串进行索引，可以使用下面的类型：varchar_pattern_ops、bpchar_pattern_ops 和 name_pattern_ops。

2.4.3 字符串相似比较算法levenshtein

levenshtein 是一个字符串比较算法，它能够计算一个字符串需要经过多少步才能变成另一个字符串，从而可以比较两个字符串的相似程度。变换过程中，每个被替换的、缺失的或多出来的字符算作一个步骤，所有步骤的和算作是两个字符串的距离。在 PostgreSQL 中，levenshtein()函数由 fuzzystrmatch 扩展模块提供。

假设有字符串 bat 和字符串 fads。

```
SELECT levenshtein('bat', 'fads');
```

因为 fads 与字符串 bat 相比，替换了两个字母（b=>f，t=>d），添加了一个字母（+s），每变换一次距离就会递增，所以它们间的 levenshtein 距离是 3。我们可以看到，每变换一步，字串的距离会交短，通过变换，距离最后会逐步减少到零（即两个字符串变成一样）。

```
SELECT levenshtein('bat', 'fad') fad,
```

```
 levenshtein('bat', 'fat') fat,
 levenshtein('bat', 'bat') bat;

 fad | fat | bat
-----+-----+-----
   2 |   1 |   0
```

大小写的变化也算是一个步骤，所以比较的时候最好把所有字符串转为相同的大小写。

```
SELECT movie_id, title FROM movies
WHERE levenshtein(lower(title), lower('a hard day nght')) <= 3;

 movie_id |      title
----------+-------------------
      245 | A Hard Day's Night
```

这确保只有细微的差别的字符串不会算成有很长的距离。

2.4.4 三连词

三连词（trigram）是从一个字符串中取出包含三个连续字符的词。pg_trgm 扩展模块可以将一个字符串分解为尽可能多的三连词。

```
SELECT show_trgm('Avatar');

              show_trgm
-------------------------------------
 {" a"," av","ar ",ata,ava,tar,vat}
```

找到一个匹配的字符串的过程可以简化，只需要统计匹配的三字母结构个数。匹配数最多的字符串就是最相似的。对于容忍小的拼写错误、甚至漏掉不重要的单词的搜索，这种查询方法是非常有用的。字符串越长，三字词就越多，则匹配的可能性就越大。它们很适合长度相近的电影片名这样的查询。

首先，将对电影的名字创建三连词索引（使用通用索引搜索树[Generalized Index Search Tree，GIST]，这是由 PostgreSQL 引擎提供的一个通用索引 API。

```
CREATE INDEX movies_title_trigram ON movies
USING gist (title gist_trgm_ops);
```

现在，即便在查询时带一点拼写错误，仍可得到不错的结果。

```
SELECT title
FROM movies
```

```
WHERE title % 'Avatre';
  title
---------
 Avatar
```

对于接受用户输入的字符匹配查询，三连词无需考虑复杂的通配符，是最好的选择。

2.4.5 全文检索

接下来，我们想让用户基于单词匹配进行全文检索，其中输入的单词可能会是复数。有时候，用户要通过几个记得的词搜索电影标题，Postgres 可以支持简单的自然语言查询处理。

1. TSVector 和 TSQuery

我们来查找名字中包含单词 night 和 day 的电影，这种查找是全文检索的强项，我们可以用@@这个全文查询运算符进行文本搜索。

```
SELECT title
FROM movies
WHERE title @@ 'night & day';

           title
-------------------------------
 A Hard Day's Night
 Six Days Seven Nights
 Long Day's Journey Into Night
```

尽管返回结果的单词 Day 是所有格形式，而且这两个词的顺序颠倒了，该查询返回了像《A Hard Day's Night》这样的电影名。@@运算符将电影名字段转换成 `tsvector` 结构体，并将查询转换成一个 `tsquery` 结构体。

`tsvector` 是一种数据类型，它将字符串分解成单词的数组（或向量），再在输入的查询字串里搜索这些单词；而 `tsquery` 则是基于某种语言的查询，如英语或法语。每种语言对应一个相应的字典（我们会在后面几段看到更多相关内容）。如果系统语言设置为英语，前一个查询相当于下面的语句：

```
SELECT title
FROM movies
WHERE to_tsvector(title) @@ to_tsquery('english', 'night & day');
```

你可以看一看，通过在字符串上彻底运行转换函数，向量和查询如何将值分开。

```
SELECT to_tsvector('A Hard Day''s Night'), to_tsquery('english', 'night & day');
    to_tsvector          |   to_tsquery
```

```
----------------------------+-----------------
 'day':3 'hard':2 'night':5 | 'night' & 'day'
```

tsvector 上的单词称为词素（lexeme），同时它还包含该词原来短语中的位置。

你可能已经注意到，《A Hard Day’s Night》的 tsvector 不包含单词 a，而且，所有像 a 这样的简单英文单词会在查询的时候被忽略。

```
SELECT *
FROM movies
WHERE title @@ to_tsquery('english', 'a');

NOTICE: text-search query contains only stop words or doesn't \
       contain lexemes, ignored
```

像 a 这样的常用词称为无用词（stop word），一般在执行查询时会被忽略。解析器会使用英语词典对字符串整型，将其转换成有意义的英语单元。可以在终端屏幕里查看 tsearch_data 目录下的英语无用词。

```
cat `pg_config --sharedir`/tsearch_data/english.stop
```

我们也可以从列表中删除 a，或者用其他字典（如 simple 字典），这个字典会把字符串分解成由非单词字符间隔开的词，并把它们转为小写。比较这两个向量：

```
SELECT to_tsvector('english', 'A Hard Day''s Night');
     to_tsvector
----------------------------
'day':3 'hard':2 'night':5
SELECT to_tsvector('simple', 'A Hard Day''s Night');
          to_tsvector
----------------------------------------
'a':1 'day':3 'hard':2 'night':5 's':4
```

使用 simple 字典可以查询包含词素 a 的任何电影。

2. 其他语言

因为 Postgres 在这里进行着某种自然语言处理，所以只有用不同的语言配置对应不同的语言，才有可能进行这样的处理。下面的命令可以查看所有已安装的语言配置：

```
book=# \dF
```

Postgres 利用字典生成 tsvector 词素，除了利用无用词，还用到其他前面没有提到的称为解析器（parser）和模板（template）的分词规则，可以通过下面的命令查看系统的规则列表：

```
book=# \dFd
```

还可以直接调用 ts_lexize()函数来对任何字典进行测试。下面的例子中，我们找到字符串 Day's 的英语词干。

```
SELECT ts_lexize('english_stem', 'Day''s');

 ts_lexize
-----------
 {day}
```

最后，前面的全文检索也适用于其他语言。如果你安装了德语字典，可以试试这个：

```
SELECT to_tsvector('german', 'was machst du gerade?');

  to_tsvector
--------------------
'gerad':4 'mach':2
```

因为 was（what）和 du（you）很常见，所以它们在德语字典中标记为无用词，而 machst（doing）和 gerade（now）被整型了。

3. 对词素索引

全文检索虽然功能强大。但是，如果不为相应的表建立索引，检索的速度会很慢。EXPLAIN 命令是一个功能强大的工具，通过它可以查看数据库内部对查询生成什么样的计划树。

```
EXPLAIN
SELECT *
FROM movies
WHERE title @@ 'night & day';
                              QUERY PLAN
---------------------------------------------------------------------------
Seq Scan on movies (cost=10000000000.00..10000000001.12 rows=1 width=68)
  Filter: (title @@ 'night & day'::text)
```

请注意 Seq Scan on movies 这一行。这在查询中基本上不是好兆头，因为它意味着读取表的每一行。因此，需要建立正确的索引。

我们将使用通用反向索引 GIN（Generalized Inverted iNdex，GIN），对我们要查询的词素创建索引，GIN 和 GIST 类似，也是索引的 API。如果你使用过像 Lucene 或 Sphinx 这样的搜索引擎，你会对*反向索引*（inverted index）比较熟悉。它是一种常见的用于对全文检索进行索引的数据结构。

```
CREATE INDEX movies_title_searchable ON movies
USING gin(to_tsvector('english', title));
```

现在有了索引，再搜索一次。

```
EXPLAIN
SELECT *
FROM movies
WHERE title @@ 'night & day';
                              QUERY PLAN
---------------------------------------------------------------------------
Seq Scan on movies (cost=10000000000.00..10000000001.12 rows=1 width=68)
  Filter: (title @@ 'night & day'::text)
```

看到了什么？什么也没变。索引是存在的，但是 Postgres 没有用它。这是因为 GIN 索引需要使用 english 的全文检索配置构造其 tsvectors 向量，但我们在查询里面没有指明该向量。现在我们需要在查询的 WHERE 子句中指定它。

```
EXPLAIN
SELECT *
FROM movies
WHERE to_tsvector('english',title) @@ 'night & day';
                                     QUERY PLAN
--------------------------------------------------------------------------------
Bitmap Heap Scan on movies (cost=4.26..8.28 rows=1 width=68)
  Recheck Cond: (to_tsvector('english'::regconfig, title) @@ '''day'''::tsquery)
  -> Bitmap Index Scan on movies_title_searchable (cost=0.00..4.26 rows=1 width=0)
       Index Cond: (to_tsvector('english'::regconfig, title) @@ '''day'''::tsquery)
```

EXPLAIN 很重要，它可以确保按你希望方式使用索引。否则，索引只是无谓的开销。

4. 发音码 metaphone

我们朝着匹配不太具体的输入迈出了一小步。LIKE 和正则表达式需要创建匹配模式，根据模式指定要求的格式准确匹配相应的字符串。levenshtein 距离可以匹配有微小拼写错误的字符串，但要求必须是非常相似的字符串。要找到合理范围内的拼写错误的匹配，三连词是很好的选择。最后，全文检索允许有自然语言的灵活性，因为它可以忽略 a 和 the 这样不重要的单词，并能处理复数形式。有时候，我们只是不知道如何正确拼写单词，但是我们知道它们的读音。

我们喜爱布鲁斯•威利斯（Bruce Willis），很想知道他参演什么样的电影。遗憾的是，我们忘了如何拼写他的名字，所以我们只能尽可能地把它拼写出来。

```
SELECT *
FROM actors
WHERE name = 'Broos Wils';
```

即使三连词在这里也用处不大（使用%而不是=）。

```
SELECT *
FROM actors
WHERE name % 'Broos Wils';
```

这里可以用（metaphone），它们是用来生成代表单词发音的发音码的算法。你可以指定输出的发音码中有多少个字符，例如，Aaron Eckhart 的 7 个字符的发音码是 ARNKHRT。

为了找到名字的发音像 Broos Wils 的人参演的所有电影，可以在查询中使用 metaphone 输出的发音码。请注意，NATURAL JOIN 是 INNER JOIN 的一种，自动以列名相同的列进行联接（例如，movies.actor_id= movies_actors.actor_id）。

```
SELECT title
FROM movies NATURAL JOIN movies_actors NATURAL JOIN actors
WHERE metaphone(name, 6) = metaphone('Broos Wils', 6);

          title
-----------------------------
 The Fifth Element
 Twelve Monkeys
 Armageddon
 Die Hard
 Pulp Fiction
 The Sixth Sense
:
```

如果你看看在线文档，会发现扩展模块 fuzzystrmatch 还有另外的函数：dmetaphone()（双发音码）、dmetaphone_alt()（替代发音）和 soundex()（19 世纪 80 年代的美国人口普查创立的、用来对比常见的美国人姓氏的古老算法）。

可以输出各函数的结果，来详细研究这些函数结果的表示方式。

```
SELECT name, dmetaphone(name), dmetaphone_alt(name),
  metaphone(name, 8), soundex(name)
FROM actors;
     name          | dmetaphone | dmetaphone_alt  | metaphone   | soundex
------------------+------------+-----------------+------------+--------
50 Cent            | SNT        | SNT             | SNT         | C530
Aaron Eckhart      | ARNK       | ARNK            | ARNKHRT     | A652
Agatha Hurle       | AK0R       | AKTR            | AK0HRL      | A236
:
```

没有哪个函数是最好的，根据你的数据集做出最优化选择。

2.4.6　组合使用字符串匹配方法

有了这些字符串检索的方法，就可以用有趣的方式组合着来用。

发音码最灵活的方面之一是其输出就是字符串。这让你能与其他字符串匹配方法混合着使用。例如，可以对 metaphone()输出结果使用三连词运算符，并按最短的 Levenshtein 距离排序结果。这意味着“找到发音像 Robin Williams 的名字，并按顺序排列。”

```
SELECT * FROM actors
WHERE metaphone(name,8) % metaphone('Robin Williams',8)
ORDER BY levenshtein(lower('Robin Williams'), lower(name));

 actor_id |     name
----------+-----------------
     4093 | Robin Williams
     2442 | John Williams
     4479 | Steven Williams
     4090 | Robin Shou
```

需要注意的是，结果并不完美。Robin Williams 排到了第三位。滥用这种灵活性可能产生其他有趣的结果，所以要小心。

```
SELECT * FROM actors WHERE dmetaphone(name) % dmetaphone('Ron');

 actor_id |   name
----------+-------------
     3911 | Renji Ishibashi
     3913 | Renée Zellweger
:
```

组合方式非常多，只受限于你的实验。

2.4.7　把电影风格表示成多维超立方体

最后介绍的扩展模块是 cube。用 cube 数据类型，将一部电影的风格类型映射到一个多维向量。然后在超级立方体内，通过一些方法，查询与给定的点最接近的点，从而给出和当前电影风格类似的所有电影的列表。

你可能注意到，在本节开始时，我们创建了一个名为 genres 的列，其数据类型是 cube，其值是 18 维空间中的一个点，每个维度代表一种风格。为什么用 *n* 维空间中的点代表电影风格？因为电影分类不是一门精确的科学，很多电影不是 100%的喜剧或悲剧，它们介乎其中。

在系统中，基于某种电影属于某种风格的强度，对电影进行评分，分数从（完全任意的数字）0 到 10，0 表示完全属于该风格，10 表示最强。

《星球大战》（Star Wars）的风格向量是(0，7，0，0，0，0，0，0，0，7，0，0，0，0，10，0，0，0)。genres 向量表里的值是向量中每个维度的位置。可以用每个 genres.position 提取 cube_ur_coord(vector，dimension)，解释它的风格值。为清楚起见，我们过滤掉得分 0 的风格。

```
SELECT name,
  cube_ur_coord('(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)', position) as score
FROM genres g
WHERE cube_ur_coord('(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)', position) > 0;
      name     | score
---------------+-------
 Adventure     |     7
  Fantasy      |     7
    SciFi      |    10
```

只要找到最近的点，我们可以找到风格相似的电影。要理解为什么这样是可行的，可以看看图 2-11。如果你最喜欢的电影是《动物之家》（Animal House），你可能喜欢的是《四十岁的老处男》（The 40 Year Old Virgin），而不是《俄狄浦斯》（Oedipus），因为它明显缺乏喜剧元素。在二维空间中，可以把寻找相似的匹配简单地转化为最近邻搜索。

还可以推广到更多的维度，代表更多的风格，如 2 维、3 维或 18 维。其原理是相同的：最近邻匹配就是风格空间中最近点，也就是最接近的风格匹配。

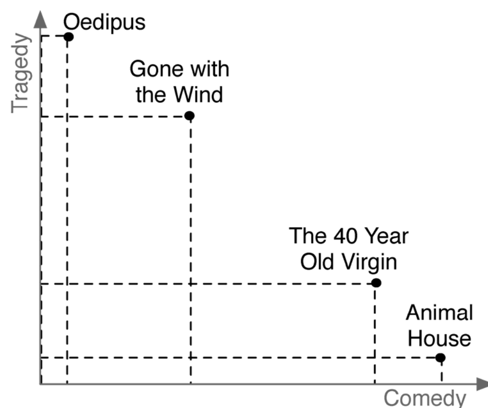


图 2-11　二维风格图中的两部电影

风格向量的最近匹配可以通过 cube_distance (point1，point2)搜索得到。这里，我们可以找到所有电影到《星球大战》的风格向量的距离，最近的排在前面。

```
SELECT *,
  cube_distance(genre, '(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)') dist
FROM movies
ORDER BY dist;
```

前面创建表时，创建了 movies_genres_cube 立方体索引。然而，即使使用索引，这个查询仍然比较缓慢，因为它需要全表扫描。它计算每一行的距离，然后将它们排序。

与其每一个点都计算距离，不如集中在一个比较可能的有界立方体（bounding cube），计算里面的点的距离。正如在找 5 个最近的镇的时候，在州的地图上找要比在世界地图上寻找更快，因为框定一个边界的话，可以减少需要看的点。

使用 cube_enlarge(cube，radius，dimensions)建立一个 18 维的立方体，其边长（半径）大于一个点。我们来看一个更简单的例子，如果围绕点(1，1)建立一个单位二维正方形，正方形的左下角是(0，0)，右上角是(2，2)。

```
SELECT cube_enlarge('(1,1)',1,2);

 cube_enlarge
---------------
 (0,0),(2,2)
```

同样的原则适用于任何维度。我们可以对一个有界的超立方框使用一个特殊的 cube 运算符“@>”，意思是包含。

下面这个查询已算出以《星球大战》（Star Wars）的风格代表的点为中心、5 单位立方体内包含的所有点的距离。

```
SELECT title, cube_distance(genre, '(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)') dist
FROM movies
WHERE cube_enlarge('(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)'::cube, 5, 18) @> genre
ORDER BY dist;
                    title                     |       dist
----------------------------------------------+------------------
 Star Wars                                    |                0
 Star Wars: Episode V - The Empire Strikes Back |              2
 Avatar                                       |                5
 Explorers                                    | 5.74456264653803
 Krull                                        | 6.48074069840786
 E.T. The Extra-Terrestrial                   | 7.61577310586391
```

可以由电影名得到风格，然后，直接执行针对该风格的计算（把子查询的结果当作一个表，并给其起个别名）。

```
SELECT m.movie_id, m.title
```

```
FROM movies m, (SELECT genre, title FROM movies WHERE title = 'Mad Max') s
WHERE cube_enlarge(s.genre, 5, 18) @> m.genre AND s.title <> m.title
ORDER BY cube_distance(m.genre, s.genre)
LIMIT 10;

 movie_id |            title
----------+----------------------------
     1405 | Cyborg
     1391 | Escape from L.A.
     1192 | Mad Max Beyond Thunderdome
     1189 | Universal Soldier
     1222 | Soldier
     1362 | Johnny Mnemonic
      946 | Alive
      418 | Escape from New York
     1877 | The Last Starfighter
     1445 | The Rocketeer
```

虽然这种电影的推荐方法并不完美，但它是一个好的开始。我们将在后面的章节，如在 MongoDB 中的二维地理搜索（参见 5.4.3 节），看到更多的维度查询。

2.4.8 第 3 天总结

今天，我们深入体验了 PostgreSQL 在字符串搜索方面的灵活性，使用了多维搜索的 cube 包。最重要的是，我们体验了 PostgreSQL 的一些非标准扩展模块，正是这些扩展让 PostgreSQL 领先于其他开源 RDBMS。还有几十（甚至几百）个扩展可供自由使用，如从地理存储到加密函数、自定义数据类型和语言扩展等。除了 SQL 核心能力之外，这些扩展模块让 PostgreSQL 光芒闪耀。

第 3 天作业

求索

1. 从官方在线文档中找到 Postgres 自带的所有可扩展包。

2. 找到网上的 POSIX 正则表达式的文档（可供后续章节中使用）。

实践

1. 创建一个存储过程，可以输入你喜欢的电影或演员的名字，它根据演员曾主演过的或类似风格的电影，返回 5 个最好的推荐。

2. 扩展电影数据库，记录用户的评论并提取关键字（除去英语的忽略字）。对比演员姓氏和相关关键字，尝试找到被谈论最多的演员。

2.5　总结

如果你没有接触过太多关系数据库的话，在决定放弃它而采用新类型的数据库之前，我们强烈建议更深入地学习 PostgreSQL，或其他关系数据库。在过去的 40 多年里，关系数据库是大量学术研究和业界改进的重点，PostgreSQL 是受益于这些进步的顶级开源关系数据库之一。

2.5.1　PostgreSQL的优点

和所有关系模型一样，PostgreSQL 的优势很多：多年的研究，几乎每个计算领域的实践使用，灵活的查询能力，非常一致和持久的数据。在大多数编程语言中，都有经过实战考验的 Postgres 驱动程序。许多编程模型，如对象关系映射（Object-Relational Mapping，ORM），假定依赖关系数据库存储。问题的关键是联接带来的灵活性。你不必知道如何针对模型进行查询，因为你总是可以执行一些联接、过滤、视图和索引，很可能总有办法提取想要的数据。

PostgreSQL 很适用于“Stepford 数据”（命名来自《贤妻》（The Stepford Wives），一个关于邻里的故事，在那里几乎每个人都保持风格和内容的一致），即数据同质，且数据很好遵从于结构化的数据定义。

此外，PostgreSQL 还提供一般的开源 RDBMS 产品没有的功能，例如，提供强大的约束机制。你可以写自己的语言扩展，自定义索引，创建自定义的数据类型，甚至重写对传入查询的解析。另外，其他的开源数据库可能有复杂的许可协议，但 PostgreSQL 采用的是最纯粹的开源方式。没有任何人拥有代码，任何人都可以对该项目做他们希望的任何事情（追究作者的责任除外）。开发和发布完全是社区支持的，如果你是一个自由软件的忠实支持者，或者有很长的浓密的胡子，你应该尊重他们，他们拒绝通过一个了不起的产品赚钱。



图 2-12　关于必要性

2.5.2　PostgreSQL的缺点

关系数据库是多年来最成功的数据库类型，尽管这一点无可否认，但在某些情况下，它可能不是非常适合。

对于 PostgreSQL 这样的关系数据库来说，分区不是强项。如果需要水平扩展而不是垂直扩展（多个并行的数据库而不是单个强大的机器或集群），可能最好寻找别的解决方案。如果数据要求过于灵活，不是很容易融入关系数据库严格的数据模式要求，或者不需要一个完整的数据库功能带来的开销，需要进行非常大量的键值对读写操作，或只需要存储二进制大对象数据，那么其他的数据存储技术可能更好。

2.5.3　结束语

关系数据库对于灵活查询是一个很好的选择。虽然 PostgreSQL 需要提前设计数据，但它不假设如何使用这些数据。只要数据模式设计相当规范，没有数据重复并且不存储可被计算出来的值，基本上就准备可以应付所有可能需要的查询。如果使用了合适的模块，调优好，建好索引，它只需消耗很少的资源就能惊人地处理几个 TB 的数据。最后，对于极度重视数据安全的人来说，PostgreSQL 的事务符合 ACID，确保你的提交是完全原子的、一致的、隔离的和持久的。

第 3 章

Riak

从事过建筑的人都知道，有种用来加强混凝土的钢条，称作钢筋。正如 Riak（读作”Ree-ahck”）一样，钢筋不会单独使用，它往往用于互相作用的各个部分，以使整个系统持久耐用。于是，系统中的每个组件都廉价又不起眼，但只要使用得当，就可构建起足够简单且牢固的基础结构。

Riak 是一种分布式的键-值（key-value）数据库。其中，值可以是任何类型的数据，如普通文本、JSON、XML、图片，甚至视频片段；而所有这些都可以通过普通的 HTTP 接口访问。你有什么，Riak 就能存什么。

容错是 Riak 的另一特性。服务器可在任何时刻启动或者停止，而不会引起任何单点故障。不管是增加或者移除服务器，甚至有节点崩溃（谁都不想这样），集群依然可以持续忙碌地运行。Riak 让你不再整夜无眠担心集群，某个节点失效不再是紧急事件，完全可以等到第二天早晨处理。Riak 的核心开发者 Justin Sheehy 曾提到：“（Riak 团队）非常注重可写入性……为的是可以回家睡觉。”

然而万事都有利弊取舍，Riak 的灵活性自有其代价。对于自由定义的（ad hoc）查询，Riak 缺乏有力支持；而键-值存储的设计，使得数据值无法相互连接（即，Riak 没有外键）。Riak 试图将这些问题各个击破，我们会在后面几天读到相关内容。

3.1 Riak 喜欢 Web

在本书中，我们会看到，比起其他数据库，Riak 更喜爱万维网（尽管 CouchDB 以很小的差距排在第二位）。可以通过 URL、HTTP 头和 HTTP 方法（如 POST）查询，Riak 则会返回相应的数据和 HTTP 状态码。

> **Riak 和 cURL**
> 本书旨在研究 7 种数据库及其概念，而非讲授新的编程语言，所以我们尽可能避免引入任何新语言。Riak 提供 HTTP REST 接口（REpresentational State Transfer，表述性状态转移），所以我们可以通过 URL 工具 cURL 与 Riak 交互。当然，在实际生产中，根据所选择的编程语言，你总会使用对应的驱动程序。而 cURL 可以让我们不依赖任何驱动或者编程语言，一览 Riak 底层 API。

像 Amazon 这样的数据中心，必须快速响应处理大量请求，Riak 是这些数据中心很棒的选择。如果客户每等待一毫秒，都会造成潜在的损失，Riak 几乎是这种案例的最佳解决方案——它易于管理、便于搭建，还能根据需要扩展。你要是用过 SimpleDB 或者 S3 之类的 Amazon Web 服务，一定会注意到 Riak 的形式与功能，和它们有相似的感觉。这可不是什么巧合，Riak 的灵感正是来自于 Amazon Dynamo 的论文。[1]

本章会探究 Riak 如何存储并检索数据，以及如何使用链接将数据捆绑。然后会探索本书中大量使用的数据检索概念：映射-归约（mapreduce）。此外，也会看到 Riak 如何把节点服务器组成集群，并且在节点发生故障时处理请求。最后，我们看看 Riak 如何处理因写入分布式服务器而产生的冲突，以及基本服务器的一些扩展。

3.2　第 1 天：CRUD、链接和 MIME

可以下载和安装 Basho[2]（资助 Riak 开发的公司）提供的 Riak，不过我们更倾向于通过编译进行安装，以便学习一些预配置的例子。若着实不想编译 Riak，那就安装预编译版本，而后下载源代码并解压到用于学习的服务器。另外，Riak 依赖于 Erlang[3]，必须安装（R14B03 及更新的版本）。

由源代码编译 Riak，需要 3 件东西：Erlang、源代码和诸如 Make 的 UNIX 编译工具。安装 Erlang 非常简单（见第 6 章），花些时间罢了。Riak 的源代码可以从代码库下载（Basho 网站上有链接——要是你没有安装 Git 或 Mercurial，下载源代码压缩包也无妨）。本章所有的例子都基于 1.0.2 版本的 Riak。

Riak 的创造者像圣诞老人一样，在新用户的长袜里放入一个很酷的玩具。在编译 Riak 的目录中运行这个命令：

```
$ make devrel
```

---

[1] http://allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
[2] http://www.basho.com/
[3] http://www.erlang.org/

命令完成后，我们可以发现三个示例服务器。启动它们：

```
$ dev/dev1/bin/riak start
$ dev/dev2/bin/riak start
$ dev/dev3/bin/riak start
```

如果某个服务器因为端口被占用而无法启动，不要慌。可以编辑故障服务器的 `/etc/app.config` 文件，修改类似如下一行内容，以改变 dev1 dev2 或 dev3 端口：

```
{http, [ {"127.0.0.1", 8091 } ]}
```

此刻，有三个名为 `beam.smp` 的 Erlang 进程在运行，分别代表各个独立的 Riak 节点（服务器实例），而不知道互相之存在。下一步是创建集群，需要用各个节点服务器的 riak-admin 命令 `cluster join`，将它们指向其他集群节点，从而相互连接。

```
$ dev/dev2/bin/riak-admin cluster join dev1@127.0.0.1
```

在 Riak 中，所有节点无主次之分，因此可以让节点服务器指向任何其他节点。既然 dev1 和 dev2 都在集群中，节点 dev3 可以指向两者中的任意一个。

```
$ dev/dev3/bin/riak-admin cluster join dev2@127.0.0.1
```

用 Web 浏览器打开 `http://localhost:8091/stats`，查看节点服务器的统计信息，可以确认它们处于正常状态。在这个过程中，会提示你下载文件，其中包含关于集群的大量信息。文件内容大致如下所示（为方便阅读，排版有所修改）：

```
{
  "vnode_gets":0,
  "vnode_puts":0,
  "vnode_index_reads":0,
  ...
  "connected_nodes":[
    "dev2@127.0.0.1",
    "dev3@127.0.0.1"
  ],
  ...
  "ring_members":[
    "dev1@127.0.0.1",
    "dev2@127.0.0.1",
    "dev3@127.0.0.1"
  ],
    "ring_num_partitions":64,
    "ring_ownership":
```

```
        "[{'dev3@127.0.0.1',21},{'dev2@127.0.0.1',21},{'dev1@127.0.0.1',22}]",
    ...
}
```

通过端口 8092（对应 dev2）和 8093（对应 dev3）访问其他节点的统计信息，不难发现，环（ring）中的所有节点服务器都是平等的参与者。我们不妨以 dev1 为例进一步了解节点统计信息。

ring_members 属性包含所有节点的名字，因此对于每个服务器都是相同的。再看 connected_nodes 属性，它则包含环中的其他所有节点服务器的列表。

显然，可以停止某个节点，从而改变 connected_nodes 属性的值...

```
$ dev/dev2/bin/riak stop
```

刷新`/stats` 页面，会发现节点 `dev2@127.0.0.1` 从 `connected_nodes` 列表中消失。启动 `dev2`，它会再次加入 Riak 环（环的内容，我们将在第 2 天讨论）。

3.2.1 REST 是最棒的（或用 cURL 时）

REST 是 REpresentational State Transfer 的缩写，意为表述性状态转移。听起来像个拗口的专业术语，但它已成为 Web 应用架构的事实标准，值得我们了解。REST 是将资源映射到 URL 及使用 CRUD 方法与这些 URL 交互的的标准。CRUD 方法意为：POST（Create）、GET（Read）、PUT（Update），以及 DELETE（Delete）。

如果还没有安装它，就安装 HTTP 客户端程序 CURL。HTTP 客户端程序 cURL 易于指定 HTTP 方法（如 GET 与 PUT）和 HTTP 头信息（如 `Content-Type`），所以不妨使用从 cURL 作为 REST 接口。利用 curl 命令，无需交互式控制台或者 Ruby 驱动程序，直接同 Riak 服务器的 HTTP REST 接口通信。

可以 ping 某个节点，以验证 `curl` 命令与 Riak 交互良好。

```
$ curl http://localhost:8091/ping
OK
```

发送一个会导致问题的请求。参数`-I` 的意思是我们只要 HTTP 响应的头。

```
$ curl -I http://localhost:8091/riak/no_bucket/no_key
HTTP/1.1 404 Object Not Found
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)
Date: Thu, 04 Aug 2011 01:25:49 GMT
```

```
Content-Type: text/plain
Content-Length: 10
```

Riak 采用 HTTP 的 URL 与方法，同样它也使用 HTTP 头和错误代码。这里的 404 响应与你平日所遇到的网页无法找到的错误代码并无不同。不妨更进一步，试试对 Riak 提交 PUT 方法请求。

参数-X PUT 意为执行 HTTP 的 PUT 方法，以存储并检索某个显式键。-H 参数将紧接其后的文本作为 HTTP 头信息。在这条命令里，还把 MIME 内容类型设置为 HTML。-d 参数后的所有内容（也就是 HTTP 体）会被 Riak 设置为一个新值。

```
$ curl -v -X PUT http://localhost:8091/riak/favs/db \
  -H "Content-Type: text/html" \
  -d "<html><body><h1>My new favorite DB is RIAK</h1></body></html>"
```

命令完毕后，如果你在浏览器中浏览 http://localhost:8091/riak/favs/db，就会看到一条出自你手的消息。

3.2.2 将值放于桶中

Riak 是一种键-值存储方式，自然需要传给它键以检索值。为了避免键冲突，Riak 将键分为各个种类，放入桶（bucket）中。比如，表示编程语言的 java 不会与作为饮料的 animals 相冲突。

我们将建立一个系统，来管理小狗旅馆中动物的信息。首先，新建一个 animals 桶，包含每个毛茸茸的房客的信息。URL 遵循这样的模式：

```
http://SERVER:PORT/riak/BUCKET/KEY
```

把数据填入 Riak 桶的一个简单方式是事先知道键。首先将小狗 Ace 加入。Ace 的小名是 The Wonder Dog，给它的键为 ace，值为{"nickname" : "The Wonder Dog", "breed" : "German Shepherd"}。你不必显示地新建桶，事实上，只要把第一个值加入某个桶名，相应的桶就创建了。

```
$ curl -v -X PUT http://localhost:8091/riak/animals/ace \
  -H "Content-Type: application/json" \
  -d '{"nickname" : "The Wonder Dog", "breed" : "German Shepherd"}'
```

加入新值会得到 204 响应代码。因为 curl 命令中的-v 参数，该命令输出了响应的头。

```
< HTTP/1.1 204 No Content
```

可以用如下命令查看已创建的桶列表。

```
$ curl -X GET http://localhost:8091/riak?buckets=true
{"buckets":["favs","animals"]}
```

当然，也可以选择在HTTP响应中包含体，而不只是头。通过增加另一个动物房客Polly，来看看命令的效果：

```
$ curl -v -X PUT http://localhost:8091/riak/animals/polly?returnbody=true \
  -H "Content-Type: application/json" \
  -d '{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}'
```

这次你会看到响应代码 200。

```
< HTTP/1.1 200 OK
```

若我们不讲究键名，Riak 会在 POST 方式的请求下生成一个键。

```
$ curl -i -X POST http://localhost:8091/riak/animals \
  -H "Content-Type: application/json" \
  -d '{"nickname" : "Sergeant Stubby", "breed" : "Terrier"}'
```

生成的键会显示在响应头的 Location 属性里，我们也能在响应头里看到成功响应代码 201。

```
HTTP/1.1 201 Created
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)
Location: /riak/animals/6VZc2o7zKxq2B34kJrm1S0ma3PO
Date: Tue, 05 Apr 2011 07:45:33 GMT
Content-Type: application/json
Content-Length: 0
```

而通过 GET 方法（如果没有指定，该请求是 CURL 的默认值）对这个位置的请求会检索这个值。

```
$ curl http://localhost:8091/riak/animals/6VZc2o7zKxq2B34kJrm1S0ma3PO
```

DELETE 方法会把这个值删除。

```
$ curl -i -X DELETE http://localhost:8091/riak/animals/6VZc2o7zKxq2B34kJrm1S0ma3PO
HTTP/1.1 204 No Content
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)
```

```
Date: Mon, 11 Apr 2011 05:08:39 GMT
Content-Type: application/x-www-form-urlencoded
Content-Length: 0
```

DELETE 方法不会返回任何响应体，但是对于成功的情况会返回响应代码 204；否则，就返回 404。

若你忘记桶中的键，可以用参数 keys=true 来查询。

```
$ curl http://localhost:8091/riak/animals?keys=true
```

当然，你也能使用参数 keys=stream，以流的形式获取数据。这对于大数集的情况更为安全适用——在数据流中，键数组的对象不断发送，直到以空数组结尾。

3.2.3 链接

将一个键关联到其他键的元数据称为链接，其基本结构如下：

```
Link: </riak/bucket/key>; riaktag=\"whatever\"
```

在尖括号（<...>）里的是值链接到的键，紧接着是一个分号，以及一个描述链接如何关联到这个值的标签（它可以是任何字符串）。

1. *链接遍历*

小狗旅馆有若干笼舍（当然是宽敞、舒适、人性化的笼舍）。为了记录某个动物在哪个笼舍，需要用到链接。通过将 cage 1 链接到 Polly 的键，可以表示 cage 1 里住着 Polly（这也会创建名为 cages 的桶）。由于这个笼舍安置在 room 101，因此对于这样的信息，将它设置为 JSON 数据。

```
$ curl -X PUT http://localhost:8091/riak/cages/1 \
  -H "Content-Type: application/json" \
  -H "Link: </riak/animals/polly>; riaktag=\"contains\"" \
  -d '{"room" : 101}'
```

不难发现，这个链接关系是单向的。事实上，才创建的这个笼舍知道 Polly 住在里面，但 Polly 的信息没有任何修改。可以这样验证，先获取 Polly 的数据，然后检查 Link 头中否有变化。

```
$ curl -i http://localhost:8091/riak/animals/polly

HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=
```

```
Vary: Accept-Encoding
Server: MochiWeb/1.1 WebMachine/1.9.0 (participate in the frantic)
Link: </riak/animals>; rel="up"
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT
ETag: "VD0ZAfOTsIHsgG5PM3YZW"
Date: Tue, 13 Dec 2011 17:54:51 GMT
Content-Type: application/json
Content-Length: 59

{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}
```

可以用逗号分隔，按需创建多个链接元数据。把 Ace 放在 cage 2 中，同时用标签 next_to 把它指向 cage 1，表示它们是相邻的两个笼舍。

```
$ curl -X PUT http://localhost:8091/riak/cages/2 \
-H "Content-Type: application/json" \
-H "Link:</riak/animals/ace>;riaktag=\"contains\",
-H "Link:</riak/cages/1>;riaktag=\"next_to\"" \
-d '{"room" : 101}'
```

Riak 链接的特别之处在于链接遍历（以及一种更强大的变形，链接 mapreduce 查询，明天会研究这部分内容）。通过在 URL 后加上结构如`/_,_,_`的链接规范，可以获取链接的数据。URL 中的下划线（_）表示链接的每个条件查询的通配符：桶（bucket）、标签（tag）和保留（keep）。这些术语稍候解释，先检索来自 cage 1 的所有链接。

```
$ curl http://localhost:8091/riak/cages/1/_,_,_

--4PYi9DW8iJK5aCvQQrrP7mh7jZs
Content-Type: multipart/mixed; boundary=Av1fawIA4WjypRlz5gHJtrRqklD

--Av1fawIA4WjypRlz5gHJtrRqklD
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=
Location: /riak/animals/polly
Content-Type: application/json
Link: </riak/animals>; rel="up"
Etag: VD0ZAfOTsIHsgG5PM3YZW
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT

{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}
--Av1fawIA4WjypRlz5gHJtrRqklD--

--4PYi9DW8iJK5aCvQQrrP7mh7jZs—
```

命令的返回结果为 `multipart/mixed` 转储的 HTTP 响应头和体，其中响应体包含所有链接的键/值。这看起来真够呛。好在明天我们会用更为强大的方法来获取链接遍历数据，它的返回值更容易阅读。但今天，我们将继续深究返回结果的语法。

你若不熟悉阅读 `multipart/mixed` 的 MIME 类型，可以先看属性 `Content-Type`，它描述了标志 HTTP 头和数据体的开始与结束边界的边界字符串。

```
--BcOdSWMLuhkisryp0GidDLqeA64
some HTTP header and body data
--BcOdSWMLuhkisryp0GidDLqeA64–
```

在这个例子中，返回的是 cage 1 所链接的数据：Polly Purebred。你可能已经注意，返回的响应头并未显示链接信息。不用紧张，数据依然存于链接到的键中。

在使用链接遍历的时候，用具体的值替代下划线进行过滤，只返回我们想要的结果。cage 2 有两个链接，因此使用链接规范进行查询会返回包含于 cage 2 中的 Ace，以及与 cage 2 相邻的 cage 1。用桶名 `animals` 替代第一条下划线，可以只检索与桶 `animals` 相关的结果。

```
$ curl http://localhost:8091/riak/cages/2/animals,_,_
```

或者通过 `tag` 条件查询，找到相邻的笼舍。

```
$ curl http://localhost:8091/riak/cages/2/_,next_to,_
```

最后那条下划线——`keep`——可以填入一个 1 或者 0。在检索两级链接或者链接的链接时（只要在一个链接模式后添加另一个链接模式），`keep` 会起作用。不妨，先通过链接 `next_to`，取得与 cage 2 相邻的键——cage 1。然后查询链接到 cage 1 的 animals。由于 `keep` 设置为 0，因此 Riak 不会返回中间步骤的数据（cage 1）。只有 Polly 的信息会返回，Polly 紧挨着 Ace 的笼舍。

```
$ curl http://localhost:8091/riak/cages/2/_,next_to,0/animals,_,_

--6mBdsboQ8kTT6MlUHg0rgvbLhzd
Content-Type: multipart/mixed; boundary=EZYdVz9Ox4xzR4jx1I2ugUFFiZh

--EZYdVz9Ox4xzR4jx1I2ugUFFiZh
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=
Location: /riak/animals/polly
Content-Type: application/json
Link: </riak/animals>; rel="up"
Etag: VD0ZAfOTsIHsgG5PM3YZW
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT

{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}
--EZYdVz9Ox4xzR4jx1I2ugUFFiZh--

--6mBdsboQ8kTT6MlUHg0rgvbLhzd--
```

如果希望得到 Polly 以及 cage 1 的信息，把 keep 设置为 1 即可。

```
$ curl http://localhost:8091/riak/cages/2/_,next_to,1/_,_,_

--PDVOEl7Rh1AP90jGln1mhz7x8r9
Content-Type: multipart/mixed; boundary=YliPQ9LPNEoAnDeAMiRkAjCbmed

--YliPQ9LPNEoAnDeAMiRkAjCbmed
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRKY+VIYo35gRfFgA=
Location: /riak/cages/1
Content-Type: application/json
Link: </riak/animals/polly>; riaktag="contains", </riak/cages>; rel="up"
Etag: 6LYhRnMRrGIqsTmpE55PaU
Last-Modified: Tue, 13 Dec 2011 17:54:34 GMT

{"room" : 101}
--YliPQ9LPNEoAnDeAMiRkAjCbmed--

--PDVOEl7Rh1AP90jGln1mhz7x8r9
Content-Type: multipart/mixed; boundary=GS9J6KQLsI8zzMxJluDITfwiUKA

--GS9J6KQLsI8zzMxJluDITfwiUKA
X-Riak-Vclock: a85hYGBgzGDKBVIcypz/fvrde/U5gymRMY+VwZw35gRfFgA=
Location: /riak/animals/polly
Content-Type: application/json
Link: </riak/animals>; rel="up"
Etag: VD0ZAfOTsIHsgG5PM3YZW
Last-Modified: Tue, 13 Dec 2011 17:53:59 GMT

{"nickname" : "Sweet Polly Purebred", "breed" : "Purebred"}
--GS9J6KQLsI8zzMxJluDITfwiUKA--

--PDVOEl7Rh1AP90jGln1mhz7x8r9--
```

在取得最终结果过程中的所有对象，都会由这条命令返回。换句话说，保留每个步骤的结果。

2. 链接之外

有了链接，可以使用 X-Riak-Meta-头前缀来存储任意元数据。如果我们想记录笼舍的颜色，而这并非日常笼舍管理中的必要数据，这时不妨通过链接把 cage 1 标记为粉色。

```
$ curl -X PUT http://localhost:8091/riak/cages/1 \
  -H "Content-Type: application/json" \
  -H "X-Riak-Meta-Color: Pink" \
  -H "Link: </riak/animals/polly>; riaktag=\"contains\"" \
  -d '{"room" : 101}'
```

使用 curl 的-I 标志获取 URL 的头，会使该命令返回元数据的名字与值。

3.2.4 Riak 的 MIME 类型

Riak 将任何数据都另存为二进制编码的值，与普通的 HTTP 并无不同。MIME 类型的意义在于赋予二进制数据上下文——目前为止，我们只处理了普通文本类型。MIME 类型存储于 Riak 服务器上，而对于客户端，它们实际上则是一个标志，以便客户端下载二进制数据后，它知道渲染成何种类型。

若想小狗旅馆记录房客的照片，我们只须使用 curl 命令的 data-binary 标志，将照片上传到服务器并指定 MIME 类型为 image/jpeg。此外，在这个照片资源上增加指向 /animals/polly 的链接，以便在取到照片时知道照片的主人。

首先，要创建名为 polly_image.jpg 的照片，在 curl 命令中指定放置照片的目录，格式与之前所执行的 curl 命令类似。

```
$ curl -X PUT http://localhost:8091/riak/photos/polly.jpg \
  -H "Content-Type: image/jpeg" \
  -H "Link: </riak/animals/polly>; riaktag=\"photo\"" \
  --data-binary @polly_image.jpg
```

在 Web 浏览器里访问 URL，正如我们平时所认为的客户端-服务器工作方式，照片发送到浏览器并正常渲染。

```
http://localhost:8091/riak/photos/polly.jpg
```

由于已经把照片链接到/animals/polly，我们可以从照片的键链接遍历到 Polly，但反过来是不行的。与关系数据库不同，这里没有关于链接的“has a”或者“is a”之类的规则。按需建立某个方向的链接。如果用例需要通过 animals 桶访问照片数据，则应该用这个方向的链接进行替代（或者追加）。

3.2.5 第 1 天总结

我们希望你有看到 Riak 作为灵活存储方案的一丝潜力。到目前为止，我们只介绍了标准键值存储的实践，以及一些链接的知识。当设计 Riak 的模式时，考虑介于缓存系统与 PostgreSQL 之间。你将数据分为不同的逻辑分类（桶），而值之间是相互关联的。然而，你不会像关系数据库一样，进一步规范化数据库的精细组件，这是因为 Riak 不在意关系联接进行值的重构。

第 1 天作业

求索

1．把 Riak 项目的在线文档加入书签，并找到 REST API 的文档。

2．找到浏览器所支持 MIME 类型的列表，尽可能完整。

3．阅读 Riak 示例配置文件 `dev/dev1/etc/app.config`，并与其他 dev 配置进行比较。

实践

1．使用 `PUT` 方法，更新 `animals/polly`，使其链接到 `photos/polly.jpg`。

2．`POST` 我们尚未尝试的 MIME 类型的文件（如 `application/pdf`），找到生成的键并在 Web 浏览器里访问相应的 URL。

3．建立一个名为 medicines 的新桶，`PUT` 一张 JPEG 图片（以合适的 MIME 类型），其键为 antibiotics，并链接到 Ace（可怜多病的小狗）。

3.3　第 2 天：Mapreduce 和服务器集群

今天我们会深入 mapreduce 框架，执行比标准键-值范式更强大的查询。然后，通过引入 mapreduce 的链接遍历，实现更强的功能。最后，我们会研究 Riak 的服务器架构，以及如何使用新的服务器布局提供灵活的一致性与可用性，即使面对网络分区依然有效。

3.3.1　填充脚本

本节需要多一点的数据。为此，需要使用不同种类的旅馆作为示例，以人类旅馆代替小狗旅馆。一个用 Ruby 写的快速填充脚本，会为 10 000 间客房的旅馆创建海量数据。

你熟悉 Ruby 吗？它是一种流行的编程语言。如果你想以简单易读的方式快速开发脚本，Ruby 十分适用。你可以在 Dave Thomas 和 Andy Hunt 所著的《Programming Ruby: The Pragmatic Programmer’s Guide》中学到更多 Ruby 的知识，当然，网上也有很多关于 Ruby 的学习资源。[1]

需要安装名为 RubyGems 的 Ruby 包管理器。[2]有了 Ruby 和 RubyGems，再安装 Riak

[1] http://ruby-lang.org
[2] http://rubygems.org

驱动程序。[1]此外，可能还需要 JSON 驱动程序，同时运行两个驱动程序以确保无误。

```
$ gem install riak-client json
```

旅馆每个房间的大小是随机的，可供 1～8 个人居住；房型亦是随机的，比如，单人间或者套房。

```
riak/hotel.rb
# generate loads and loads of rooms with random styles and capacities
require 'rubygems'
require 'riak'
STYLES = %w{single double queen king suite}

client = Riak::Client.new(:http_port => 8091)
bucket = client.bucket('rooms')
# Create 100 floors to the building
for floor in 1..100
  current_rooms_block = floor * 100
  puts "Making rooms #{current_rooms_block} - #{current_rooms_block + 100}"
  # Put 100 rooms on each floor (huge hotel!)
  for room in 1...100
    # Create a unique room number as the key
    ro = Riak::RObject.new(bucket, (current_rooms_block + room))
    # Randomly grab a room style, and make up a capacity
    style = STYLES[rand(STYLES.length)]
    capacity = rand(8) + 1
    # Store the room information as a JSON value
    ro.content_type = "application/json"
    ro.data = {'style' => style, 'capacity' => capacity}
    ro.store
  end
end

$ ruby hotel.rb
```

我们现在已经为人类旅馆（不是小狗旅馆了）填充了示例数据，然后可以在此之上进行 mapreduce 操作。

3.3.2　mapreduce 介绍

作为在多个节点上执行并行任务的算法框架，mapreduce 的普及是 Google 对计算机科

[1] http://rubygems.org/gems/riak-client

学最大的持久贡献之一。mapreduce 最早在 Google 相关主题的开创性论文中描述[1]，并成为分区容忍性数据存储库中，执行自定义查询的有用工具。

mapreduce 会把问题分解为两部分。一，通过 `map()`方法，将一列数据转换成另一不同类型的一列数据。二，通过 `reduce()`函数，将 `map()`方法所生成的那列数据转换成一个或者多个标量值。这种模式允许系统将任务分成更小的组件任务，然后跨大规模集群服务器并行运行这些任务。通过将包含`{country : 'CA'}`的 Riak 值映射到`{count : 1}`，然后计算所有这些 count 的数目以进行归约，就能算出包含`{country : 'CA'}`的 Riak 值的总数。如果在数据集中有 5012 个加拿大人，归约的结果则是`{count : 5012}`。

```
map = function(v) {
  var parsedData = JSON.parse(v.values[0].data);
  if(parsedData.country === 'CA')
    return [{count : 1}];
  else
    return [{count : 0}];
}

reduce = function(mappedVals) {
  var sums = 0;
  for (var i in mappedVals) {
    sums[count] += mappedVals[i][count];
  }
  return [{count:sums}];
}
```

在某种程度上，mapreduce 与通常运行查询的方法相反。运行在 Rails 系统上的 Ruby 脚本能以如下方式抓取数据（通过它的 ActiveRecord 接口）：

```
# Construct a Hash to store room capacity count keyed by room style
capacity_by_style = Hash.new{|h.k| h[k]=0}
Room.find_each do |room|#all and each is redlly bad
# wouldn't consider it fair to AR
# in this context
  capacity_by_style[room.style] += room.capacity
end
```

`Room.all` 对后台数据库执行 SQL 查询，类似这样：

```
SELECT * FROM rooms;
```

数据库把所有的结果发送到应用服务器，而后应用服务器代码对这些数据执行某些操

[1] http://labs.google.com/papers/mapreduce.html

作。在这个用例中，遍历旅馆中的每个客房，然后为每种房型的客房计算总容量（比如，旅馆中所有套房的容量为 448 个房客）。这对于小数据集差强人意。但是，随着客房数目的增长，由于数据库持续将每个客房的数据串流到应用服务器，系统会渐渐变慢。

mapreduce 则以相反的方式运作。可以这样理解，在常规查询方式中，首先从数据库抓取数据，然后客户端（或者应用服务器）获取并处理数据，而 mapreduce 作为一种工作模式，会将某种算法传给数据库节点，之后每个节点负责返回各自的查询结果。节点服务器上的每个对象都“映射”（mapped）到一些常用的键，这些键用来将数据分组，接着，所有相互匹配的键都“归约”（reduced）成某些单一的值。

对于 Riak，这意味着是数据库服务器负责映射并归约每个节点上的值。归约的这些值在网络上传输，而在某个其他服务器上（往往是发出请求的服务器）会进一步归约这些值，直到最终的结果传递给发出请求的客户端（视情况而定，或者传递给 Rails 应用服务器）。

这个简单的变换造就了一种强大的方法，复杂的算法得以在每个节点服务器上运行，从而计算得到传输开销很小的结果，并返回给发送请求的服务器。算法先发送到数据，再将数据发送到算法，是一种更快的查询方式。在图 3-1 中，我们看到如何计算一桶电话账单的总费用，电话账单以电话号码为键，分布在 3 个服务器上，每个服务器存有前缀相类似的所有电话号码。

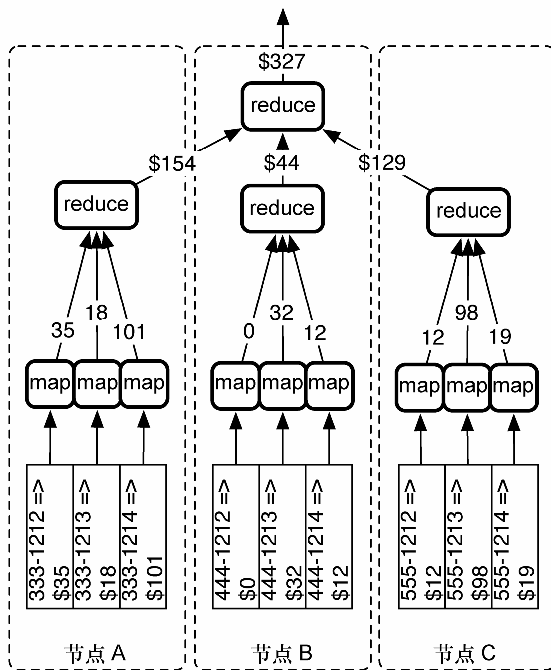


图 3-1 map 函数输出给 reduce 函数再输出给其他 reduce 函数

map 函数的结果会输入 reduce 函数；然而，map 函数和 reduce 函数共同协作的结果也会注入其后连续调用的 reduce 函数。之后的章节会再次讨论这种思想，因为对于编写有效的 mapreduce 查询来说，这是其中重要而微妙的一部分。

3.3.3 Riak 中的 mapreduce

让我们试着为 Riak 数据集创建 mapreduce 函数，该函数的功能就如之前所讨论的，计算旅馆的容量。运用于 Riak 的 mapreduce 有一个实用的功能，可以单独运行 `map()`函数，查看所有的中间结果（所谓中间结果，假设你会继续运行 reduce 函数）。我们不妨放缓速度，先看看 101、102 和 103 房间的查询结果。

实现映射依赖于我们正在使用的编程语言和源代码；之后才能实际编写 map 函数，我们所用的是 JavaScript（JavaScript 实现的这个函数只是一个字符串，所以始终需要将某些字符转义）。

在 cURL 中使用`@-`命令可以使控制台标准输入保持打开状态，直至接收到 `CTRL+D`。命令中的数据会注入 HTTP 请求体，并以 POST 方式提交给命令`/mapred`（注意，这里的 URL 是`/mapred`，而非`/riak/mapred`）

```
$ curl -X POST -H "content-type:application/json" \
http://localhost:8091/mapred --data @-
{
  "inputs":[
    ["rooms","101"],["rooms","102"],["rooms","103"]
  ],
  "query":[
    {"map":{
      "language":"javascript",
      "source":
        "function(v) {
          /* From the Riak object, pull data and parse it as JSON */
          var parsed_data = JSON.parse(v.values[0].data);
          var data = {};
          /* Key capacity number by room style string */
        data[parsed_data.style] = parsed_data.capacity;
        return [data];
      }"
    }}
  ]
}
CTRL-D
```

/mapred 命令希望得到有效的 JASON，这里规定了 mapreduce 命令的格式。首先，选择三个房间，具体的做法是设置一个数组作为输入值“inputs”，其中包含`[bucket，key]`

桶键组合。不过，这条命令的真正重点在于 query 的值，它可以接受的内容有，包含其他对象的 JSON 对象的数组，如 map、reduce 和/或 link（后面有更多关于 link 的讨论）。

这一切归根结底取决于 `v.values[0].data`，它会被 `JSON.parse（...）`解析为 JSON 对象，然后返回以房型（`parsed_data.style`）为键的房间容量（`parsed_data.capacity`）。你能得到的结果如下所示：

```
[{"suite":6},{"single":1},{"double":1}]
```

这只是来自于 101、102 和 103 房间的三个对象的 JSON 数据。

不一定要把数据简单地输出为 JSON。事实上，可以将每个键及其值转换为我们希望的任何形式。这里仅仅深入讨论了响应体，然而其实还取回了元数据、链接信息、键以及数据。于是，任何事情都可能发生——可以将每个键值映射到某些其他的值。

如果你觉得可以胜任，通过用 `rooms` 桶名修改输入数组[bucket，key]能返回全部 10 000 个客房的映射值，如下所示：

```
"inputs":"rooms"
```

郑重警告：这会输出大量数据。最后，值得一提的是，从 1.0 版本的 Riak 起，mapreduce 函数由称为 Riak Pipe 的子系统处理；而更早的系统则使用遗留的 mapred_system。作为最终用户，这对你不会有大影响，但这种变化着实提升了运行速度与稳定性。

1. 可存储的函数

Riak 还提供了一种选择——在桶里存储 map 函数。这是将算法移入数据库的另一个例子。它是一个存储过程，或者更确切地说，是一个用户定义的函数，这与关系数据库中使用多年的理念类似。

```
$ curl -X PUT -H "content-type:application/json" \
http://localhost:8091/riak/my_functions/map_capacity --data @-
function(v) {
  var parsed_data = JSON.parse(v.values[0].data);
  var data = {};
  data[parsed_data.style] = parsed_data.capacity;
  return [data];
}
```

只要函数安全地存于 Riak，就能指向包含这个函数的桶与键，从而运行此函数。

```
$ curl -X POST -H "content-type:application/json" \
http://localhost:8091/mapred --data @-
```

```
{
  "inputs":[
    ["rooms","101"],["rooms","102"],["rooms","103"]
  ],
  "query":[
    {"map":{
      "language":"javascript",
      "bucket":"my_functions",
      "key":"map_capacity"
    }}
  ]
}
```

正如在请求中置入 JavaScript 源代码一样，可以通过这种方式得到相同的结果，

2. 内置函数

可以使用附于 JavaScript 对象 `Riak` 之中的 Riak 内置函数。如果运行如下所示的代码，room 对象会把值映射到 JSON 并返回它们。函数 `Riak.mapValuesJson` 的作用就是将值映射为 JSON。

```
curl -X POST http://localhost:8091/mapred \
-H "content-type:application/json" --data @-
{
  "inputs":[
    ["rooms","101"],["rooms","102"],["rooms","103"]
  ],
  "query":[
    {"map":{
      "language":"javascript",
      "name":"Riak.mapValuesJson"
    }}
  ]
}
```

事实上，在一个名为 `mapred_builtins.js` 的文件中（该文件可以在线或者深入代码找到），可以找到多 Riak 所提供的更多内置函数。同样，能用这样的语法调用自己的内置函数，会在明天研究这部分内容。

3. 归约

映射（mapping）发挥着它的作用，然而只能用它把单个值转化为其他单个值。对数据集做某种分析，甚至是简单地对记录计数，都需要映射之外其他的其他步骤。而这就是归约（reducing）发挥其作用之处。

我们稍早讨论的 SQL/Ruby 例子（见 3.3.2 节）演示了如何遍历每个值，以及如何计算每种房型的总容量。下面将用 JavaScript 的 `reduce` 函数执行这个功能。

传给`/mapred` 的大部分命令是相同的。这次将增加 `reduce` 函数。

```
$ curl -X POST -H "content-type:application/json" \
http://localhost:8091/mapred --data @-
{
  "inputs":"rooms",
  "query":[
    {"map":{
      "language":"javascript",
       "bucket":"my_functions",
       "key":"map_capacity"
    }},
    {"reduce":{
      "language":"javascript",
      "source":
         "function(v) {
           var totals = {};
           for (var i in v) {
             for(var style in v[i]) {
               if( totals[style] ) totals[style] += v[i][style];
               else                totals[style] = v[i][style];
             }
           }
           return [totals];
        }"
     }}
  ]
}
```

对所有客房运行这个函数回返回以房型为键的总容量。

```
[{"single":7025,"queen":7123,"double":6855,"king":6733,"suite":7332}]
```

你在此得到的总容量与之前的结果不会完全一致，因为客房数据是随即生成的。

4. 键过滤器

Riak 最近的新增特性是称为键过滤器的概念。键过滤器是一组命令的集合，它可以在执行 mapreduce 之前预处理键。通过这种便捷方式，可以避免加载不需要的键。在如下例子中，将把每个作为键的房号转换成整数，并检查它是否小于 1000（由此只统计最低 10 层的数据；10 层以上的任何客房都会忽略）。

在返回客房容量的 mapreduce 过程中，以如下代码块（必须以逗号结尾）替代“inputs”:” rooms”。

```
"inputs":{
  "bucket":"rooms",
  "key_filters":[["string_to_int"], ["less_than", 1000]]
},
```

你应该注意到了两点：查询快了很多很多（因为只处理了需要的数据），查询得到的容量也更少（因为只计算了最低 10 层的数据）。

mapreduce 是绑定数据并对此做总体分析的强大工具。在本书中，我们将经常重温这个概念，其核心概念是完全相同的。Riak 对基本的 mapreduce 形式略作调整，即增加了链接（link）。

5. 利用 mapreduce 进行链接遍历

昨天介绍了链接遍历。今天会讨论如何利用 mapreduce 进行链接遍历。查询部分会包含 `map` 与 `reduce` 之外的另一个值选项，即链接。

让我们重回昨天小狗旅馆例子中的笼舍桶，编写一个仅返回 cage 2 的映射（记住，住着小狗 Ace 的笼舍）。

```
$ curl -X POST -H "content-type:application/json" \
http://localhost:8091/mapred --data @-
{
   "inputs":{
      "bucket":"cages",
      "key_filters":[["eq", "2"]]
   },
   "query":[
      {"link":{
         "bucket":"animals",
         "keep":false
      }},
      {"map":{
         "language":"javascript",
         "source":
            "function(v) { return [v]; }"
      }}
   ]
}
```

> **归约器模式**
>
> 如果归约器模式遵循与 map 函数一样的模式，写 reduce 函数就会容易些。也就是说，如果将单个值映射为
>
> ```
> ...[{name:'Eric', count:1}]
> ```
>
> ...那么 reduce 函数的结果应像这样:
>
> ```
> [{name:'Eric', count:105}, {name:'Jim', count:215}, …]
> ```
>
> 当然，这不是必需的；而是实践所得。归约器的输出作为其他归约器的输入，可是你不知道 map 函数的输出、reduce 函数的输出或者结合两者的输出，能否作为 `reduce` 函数的输入。然而，如果遵循相同的对象模式，你就不必在意这些；所有都会是一致的。否则，`reduce` 函数必须总是检查它收到的数据类型，并作相应的决策。

虽然对 `cage` 桶运行 mapreduce 查询，但返回的是小狗 Ace 的信息，这是因为 Ace 与 cage 2 之间存在链接。


```
[{
  "bucket":"animals",
  "key":"ace",
  "vclock":"a85hYGBgzmDKBVIsrDJPfTKYEhnzWBn6LfiP80GFWVZay0KF5yGE2ZqTGPmCLiJLZAEA",
  "values":[{
     "metadata":{
        "Links":[],
        "X-Riak-VTag":"4JVlDcEYRIKuyUhw8OUYJS",
        "content-type":"application/json",
        "X-Riak-Last-Modified":"Tue, 05 Apr 2011 06:54:22 GMT",
        "X-Riak-Meta":[]},
     "data":"{\"nickname\" : \"The Wonder Dog\", \"breed\" : \"German Shepherd\"}"
  }]
}]
```


数据与元数据（一般在 HTTP 头中返回）都会出现在值数组中。

将映射、归约、链接遍历以及键过滤器放在一起，你便能对任意 Riak 键数组执行查询。相比从客户端扫描所有数据，这种方式的效率大大提高。

由于这些查询往往在若干节点服务器上同时运行，因此你不必为此长时间等候。但是，如果你真的不能接受等待，查询还有一个选项：超时（`timeout`）。将超时设置为以毫秒为单位的值（默认值是`"timeout": 60000`，即 60 秒），如果查询在指定时间内没有完成，它就会取消。

3.3.4　关于一致性和持久性

Riak 服务器架构消除了单点故障（所有节点都是对等的），并允许任意增大或缩小集群。这在处理大规模部署时是非常重要的——即使若干节点发生故障或者没有响应，数据库仍然可用。

将数据分布于多个服务器须面对一个棘手的先天问题。如果你想要数据库在网络分区发生（即，某些消息丢失了）时依然能够运行，你必须做一个权衡。或者对服务器请求保持可用，或者拒绝请求，以保证数据的一致性。要创建一个具备完全一致性、可用性与分区容错性的分布式数据库，是不可能的。你只能保证三个中的两个（分区容错性与一致性、分区容错性与可用性，或者一致性与可用性但非分布式数据库）。这称为 CAP 定理（一致性，可用性，分区容错性，Consistency，Availability，Partition tolerance）。详见附录 2，足以说明这是系统设计中的问题。

然而，这个定理有个漏洞。现实是，在任何时刻，不能同时保证一致性、可用性与分区容错性。Riak 利用这个事实，允许在每个请求的基础上，以可用性交换一致性。我们先看看 Riak 如何将服务器组成集群，然后讨论如何调整读写来和集群交互。

1. Riak 环

Riak 将其服务器配置划分为分区，以一个 160 比特的数字（即 2^{160}）表示。Riak 团队喜欢用圆圈代表这个巨大的整数，他们称之为环。当把一个键哈希为一个分区，这个环会帮忙指向存有相应值的服务器。

在搭建一个 Riak 集群时，遇到的第一个问题就是你想要多少个分区。不妨考虑这样的案例，你有 64 个分区（Riak 的默认分区数）。如果把这 64 个分区分布在 3 个节点（或者服务器），Riak 会为每个节点分配 21 或者 22 个分区（64/3）。每个分区被为一个虚拟节点，或者 vnode。每个 Riak 服务会在启动时计数，依次清点分区直到所有的 vnode 都清点完毕，见图 3-2。

节点 A 管理 vnode 1、4、7、10…63。这些 vnode 映射到 160 个比特表示的分区。你若查看三个开发服务器的状态（记住昨天讲过的 `curl -H "Accept: text/plain" http://localhost:8091/stats`），会见到如下所示的一行：

```
"ring_ownership": \
"[{'dev3@127.0.0.1',21},{'dev2@127.0.0.1',21},{'dev1@127.0.0.1',22}]"
```

每个对象的第二个数字就是该节点所拥有的 vnode 数量。一共是 64（21+21+22）。

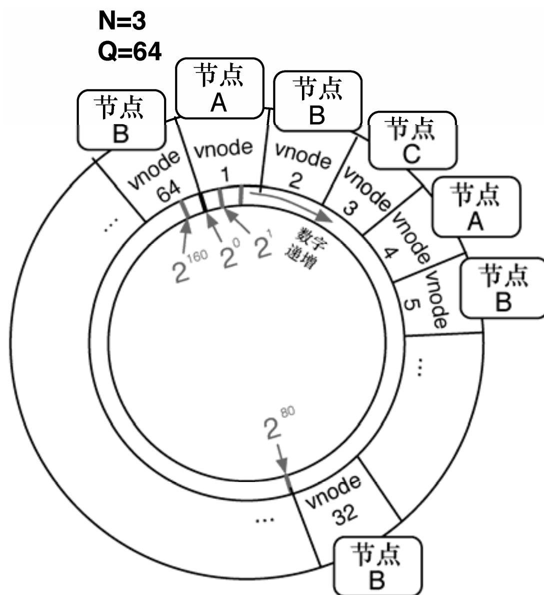


图 3-2　64 个 vnode 的‘Riak 环’，分布于 3 个物理节点

每个 vnode 代表一系列经过哈希的键。当插入键 101 的客房数据时，它会哈希到 vnode 2 的范围，于是键-值对象会存储在节点 B。这么做的好处在于，如果需要查找键存储在哪个服务器，Riak 只需要对键做哈希运算，找到对应的 vnode。具体而言，Riak 会把哈希转换成一组潜在的 vnode，并使用其中的第一个。

2. 节点/读/写

Riak 允许通过改变三个值：*N*、*W* 与 *R*，来控制集群的读写。*N* 是一次写入最终复制到的节点数量，换句话说，就是集群中的副本数量。*W* 是一次成功地写入响应之前，必须成功写入的节点数量。如果 *W* 小于 *N*，就认为某次写入是成功的，即使 Riak 依然在复制数据。最后，*R* 是成功读出一项数据所必需的节点数量。如果 *R* 比可用的复制数量大，读出请求将会失败。

我们来更详细地研究每一项。

当在 Riak 中写入对象时，可以选择在多个节点上创建这个值的副本。这么做的好处是，如果某个节点发生故障，还有另一个节点上的副本可用。如果某个值在若干节点上有副本，这些节点的数量（即 *N* 的值）以桶属性 `n_val` 表示；默认为 3。通过在 `props` 对象中设置新值，可以修改桶的属性。此处，设置 animals，使其 `n_val` 属性为 4：

```
$ curl -X PUT http://localhost:8091/riak/animals \
  -H "Content-Type: application/json" \
```

```
    -d '{"props":{"n_val":4}}'
```

N 为最终将包含正确值的节点总数。但这并不意味着在调用返回前，我们必须等待值复制到所有这些节点。有时，我们只希望客户端立刻返回，让 Riak 在后台复制。而有时，我们希望等待，直到 Riak 完成所有 *N* 个节点的副本复制，才让客户端返回。

在认为一次操作成功之前，把必须成功执行的写操作次数设置为 *W*。虽然最终会写入 4 个节点，然而，如果把 W 设置为 2，仅仅在生成两个副本之后，一次写操作就会返回。剩下的两个节点将在后台复制。

```
curl -X PUT http://localhost:8091/riak/animals \
  -H "Content-Type: application/json" \
  -d '{"props":{"w":2}}'
```

最后，还能使用 *R* 的值。一次成功的读操作之前，必须读出 R 个节点的值。可以为 *R* 设置默认值，就如之前处理 `n_val` 与 `w` 一样。

```
curl -X PUT http://localhost:8091/riak/animals \
  -H "Content-Type: application/json" \
  -d '{"props":{"r":3}}'
```

但是，Riak 还提供了更为灵活的解决方案。可以通过在每个请求的 URL 中设置参数 r，选择我们想读的节点数量。

```
curl http://localhost:8091/riak/animals/ace?r=3
```

你也许正在问自己，为何需要从多个节点读取信息。毕竟，写入的值最终会复制到 *N* 个节点，可以从其中任意一个节点读取。为说明这个问题，我们发现可视化方法更容易解释这种策略。关于高可用性的漫画见图 3-3。



图 3-3　高可用性

比如，把 NRW 设置成 `{"n_val":3，"r":2，"w":1}`，见图 3-4。这种设置使系统的写入操作响应更快，原因在于写操作返回前，只要写入一个节点即可。但是，存在一种可能，恰巧在节点同步之前，另一个读出操作执行了。即使从两个节点读出数据，也有可能得到一个旧值。

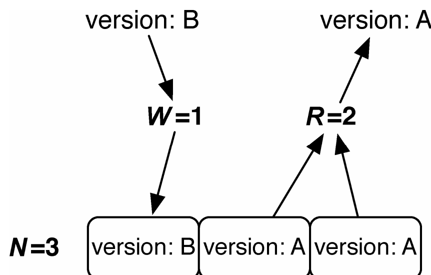


图 3-4　最终一致性：$W+R \leqslant N$

一种确保我们能够读到最新值的方法是让 $W=N$、$R=1$，就像这样：`{"n_val":3，"r":1，"w":3}`（见图 3-5）。从本质上讲，这就是关系数据库的做法；通过确保写操作在返回之前完成，以保证一致性。但是，这确实会降低写操作的性能。

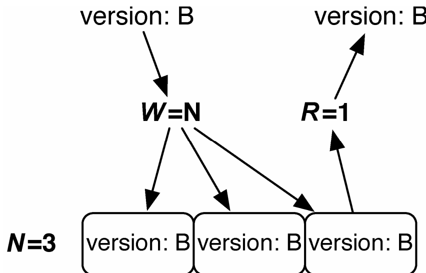


图 3-5　写操作实现一致性：$W=N$，$R=1$

或者，可以只写入单个节点，而从全部节点读出。让 $W=1$、$R=$N，就像这样：`{"n_val":3，"r":3，"w":1}`（见图 3-6）。尽管你可能因此读到一些旧值，但你也能保证检索到最新的值。这时，你必须要做的只是找出哪个是最新值（通过一个向量时钟实现，我们明天会进行讨论）。当然，这也会带来副作用，如前所示，读操作会因此减慢。

还有一种选择，可以让 $W=2$、$R=2$，就如`{"n_val":3，"r":2，"w":2}`这样（见图 3-7）。在这种方法中，只需要写入多于一半的节点，并从多于一半的节点读出，就依然可以保证一致性。同时，分担介于读写之间的延时。这称为法定数（quorum），是确保数据一致性的方法中开销最小的。

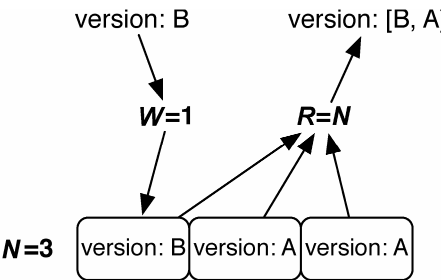


图 3-6　读操作实现一致性：W=1，R=N

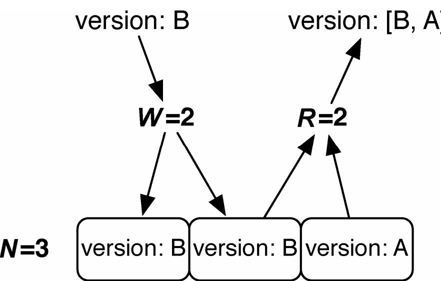


图 3-7　法定数实现一致性：W+R>N

可以把 R 或者 W 自由地设置为 1～N 之间的值，但一般会选定 1、N 或者法定数。这些是 R 和 W 可用的常见值，以字符串表示，定义见下表：

术　　语	定　　义
一	这就是值 1。设置 W 或 R 意味着只需要有一个节点响应，请求就成功了
全部	这和值 N 是一样的。将 W 或 R 设置为这个值意味着所有复制的节点必须响应
法定数	这意味着将值设置为 N/2+1。设置 W 或 R 意味着过半数节点必须响应才算成功
默认值	不管桶的 W 或 R 值设置为什么。通常默认值是 3

这些值除了作为合法的桶属性，也能将它们用做查询参数值。

```
curl http://localhost:8091/riak/animals/ace?r=all
```

从所有节点读出数据的危险在于，一旦某个节点出现故障，Riak 可能无法满足你的读请求。作为实验，关闭开发服务器 dev3。

```
$ dev/dev3/bin/riak stop
```

现在如果尝试从所有节点读取数据，请求失败的概率会很大（如果请求没有失败，可

以尝试把 dev2 也关闭，或者关闭 dev1，并从端口 8092 或者 8093 读取数据；我们无法控制 Riak 会写入哪个 vnode）。

```
$ curl -i http://localhost:8091/riak/animals/ace?r=all
HTTP/1.1 404 Object Not Found
Server: MochiWeb/1.1 WebMachine/1.7.3 (participate in the frantic)
Date: Thu, 02 Jun 2011 17:18:18 GMT
Content-Type: text/plain
Content-Length: 10

not found
```

如果你的请求无法满足，会得到 404 错误代码（未找到对象），这在 HTTP 请求的领域内，是合理的结果。无法找到检索对象的原因是，没有足够的副本以满足 URL 请求。当然，这不是什么好事，因而迫使 Riak 进行读操作修复：从依然可用的节点服务器请求键的 *N* 个副本。如果尝试再次访问同一 URL，会取到键的值而不是 404 错误。Riak 在线文档中有一个用 Erlang 实现的范例[1]。

然而，一种更可靠的做法是寻求一个法定数（从大部分而非全部 vnode 获取数据）。

```
curl http://localhost:8091/riak/animals/polly?r=quorum
```

只要在每次写操作时，强制写入法定数量的节点，就能保证读操作的一致性。另一个可以动态设置的值是 *W*。

你若不想等待 Riak 写入任何节点，不妨把 *W* 设置为 0，意思是“Riak，我相信你会将数据写入；只管返回吧”。

```
curl -X PUT http://localhost:8091/riak/animals/jean?w=0 \
-H "Content-Type: application/json"
-d '{"nickname" : "Jean", "breed" : "Border Collie"}' \
```

将这种可定制化的功能置于一旁，除非有足够好的理由，否则大部分时间你会选择使用默认设置。让 *W*=0，对日志是非常适用的配置；而让 *W*=*N* 且 *R*=1，很适合高速读取数据，极少写入数据的场景。

3. 写入与持久化写入

我们一直对你保守着一个秘密。Riak 的写入操作未必是持久化的，也就是说，数据并非立刻写入磁盘。即使一个节点的写入操作成功执行，依然可能因为故障而丢失这个节点

[1] http://wiki.basho.com/Replication.html

中的数据；就算 $W=N$，节点服务器也会发生故障，丢失数据。写入的数据在存到磁盘之前，会于内存中缓存片刻，而这毫秒的间隙正是危险的所在。

这的确是个坏消息。但好消息是 Riak 提供了名为 DW 的单独设置，用于持久化写入。在这种设置下，直到对象写入给定数量的节点上的磁盘，Riak 才会成功返回，因此，这会减慢速度，同时降低风险。这里把 dw 设置为 1，以确保至少一个节点保存数据。

```
$ curl -X PUT http://localhost:8091/riak/animals \
-H "Content-Type: application/json" \
-d '{"props":{"dw":"one"}}'
```

或者，如果你愿意，可以基于每个写操作，使用 URL 中的查询参数 dw 进行设置。

4. 关于临时转移的说明

尝试写入不可用的节点仍然会执行成功，并得到这样的返回“204 No Content”。这是因为 Riak 会把数据先写入附近的一个节点，该节点一直保存数据，直到它能将这些数据交给不可用的节点。这在短期内是一个很棒的安全网，因为一个节点服务器 一旦出现故障，另一个 Riak 节点将接管。当然，如果服务器 A 的所有请求都转发到服务器 B，服务器 B 就得处理双倍的负载。这里存在一种风险，服务器 B 可能因为高负载而出现故障，于是渐渐蔓延到服务器 C 和 D，如此类推。这称为级联故障（cascading failure），罕见但依然可能发生。不妨认为这是一个郑重的警告，不要耗尽每台 Riak 服务器的负载，说不定在某个时刻，某个节点服务器就必须填补缺口。

3.3.5　第 2 天总结

今天，我们学习了 Riak 的两大主题：强大的 mapreduce 方法和灵活的服务器集群能力。mapreduce 在本书中的许多其他数据库中也用到，所以如果你对它还有疑问，我们建议你重新阅读第 2 天的第一部分，并查看 Riak 的在线文档[1]和 Wikipedia[2]上的文章。

第 2 天作业

求索

1. 阅读关于 Riak mapreduce 的在线文档。

2. 在大量预构建的 mapreduce 函数中，找到 Riak 贡献的函数库。

---

[1] http://wiki.basho.com/MapReduce.html
[2] http://en.wikipedia.org/wiki/MapReduce

3. 从在线文档中找到键过滤器的完整列表，包括字符串转换过滤器 `to_upper`，查找数字的过滤器，甚至涉及一些简单的 Levenshtein distance[1]字符串匹配，以及逻辑操作符与/或/非。

实践

1. 针对 `rooms` 桶，编写 man 和 reduce 函数，查询每层楼客房的总容量。
2. 以过滤器扩展前面编写的函数，仅查找 42 层和 43 层的客房容量。

3.4 第 3 天：解决冲突和扩展 Riak

今天将探索 Riak 的边缘地带。我们已经看到 Riak 是一个如何简单的分布式键值数据库。在处理多个节点的时候，可能发生数据冲突，而有时，我们不得不解决它们。通过向量时钟与同级解决方案（sibling resolution），Riak 提供了一种机制，理清写操作的发生顺序，找出最近的写入操作。

我们也会看到，通过提之前/后钩子程序（pre-/post-commit hooks）如何验证传入数据的。此外，我们将用 Riak 搜索，把 Riak 扩展到我们自己的个人搜索引擎，以及通过二级索引实现更快的查询。

3.4.1 以向量时钟解决冲突

向量时钟[2]是像 Riak 这样的分布式系统所使用的令牌，用以理顺发生冲突的键值更新顺序，确保服务器上的数据正确有效。由于若干客户端可能连接到不同的服务器，并且一个客户端更新这个服务器，另一个客户端更新那个服务器（你无法控制写入哪个服务器），因此记录哪些更新以怎样的顺序发生，是非常重要的。

你或许在想“给值加上时间戳，采用时间戳最新的值即可”，但是，这种做法仅仅在所有节点服务器的时钟完全同步的集群中起效。Riak 对此不做要求，事实上，保持时钟同步是最为困难的，在许多情况下，甚至是不可能的。使用集中式的时钟系统是对 Riak 哲学的诅咒，显然，它意味着单点故障的可能性。

向量时钟通过这种方式发挥作用——对每个键值事件（创建、更新或者删除）做标记，标记包含两项内容：哪个客户端更新了数据与以哪种顺序更新。在此基础上，

[1] Levenshtein distance，即编辑距离，详见 http://en.wikipedia.org/wiki/Levenshtein_distance——译者注
[2] http://en.wikipedia.org/wiki/Vector_clock

客户端或者应用开发者，得以决定谁在冲突中胜出。如果你熟悉像 Git 或者 Subversion 之类的版本控制系统，不难发现，在解决两个人修改同一文件的版本冲突时，本质并无不同。

1. 理论中的向量时钟

既然你的小狗旅馆运作良好，你必须开始对客户有更强的选择性。为了做出最好的决策，你筹备了一个由三位动物专家组成的委员会，帮忙评判哪些新的小狗是不错的选择。委员会为每个小狗评分，分数介于 1（不够好的选择）～4（完美的候选）之间。所有委员会成员（Bob、Jane 和 Rakshith）必须完全达成一致。

每名委员把自己的客户端连接到数据库服务器，并且每个客户端会为每个请求盖上独一无二的戳——客户端 ID。客户端 ID 用来组成向量时钟，并追踪更新对象头部的客户端中最后的那个。我们来看一个简单的伪代码示例，稍后尝试 Riak 中的例子。

Bob 首先创建一个对象，其中包含名为 Bruiser 的新来小狗的分数，体面的 3 分。向量时钟为他的名字与版本进行如下编码。

```
vclock: bob[1]
value: {score : 3}
```

Jane 取到这条记录，给 Bruiser 的评分为 2。为 Jane 的更新所创建的 vclock 继承自 Bob 的 vclock，所以 Jane 的版本 1 追加到向量的最后。

```
vclock: bob[1], jane[1]
value: {score : 2}
```

同时，Rakshith 取到 Bob 所创建但 Jane 尚未更新的版本。他喜欢 Bruiser，并给了 4 分。和 Jane 一样，他的客户端名字也以版本 1，追加到向量时钟的末尾。

```
vclock: bob[1], rakshith[1]
value: {score : 4}
```

当天晚些时候，Jane（作为委员会主席）复查了评分。由于 Rakshith 的更新向量与 Jane 的更新向量并列，并未出现在 Jane 的更新向量之后，因此需要解决存在冲突的更新。因为 Jane 取到了两条记录，如何解决冲突全由她决定。

```
vclock: bob[1], jane[1]
value: {score : 2}
---
```

```
vclock: bob[1], rakshith[1]
value: {score : 4}
```

她选择折中，于是把分数更新为 3。

```
vclock: bob[1], rakshith[1], jane[2]
value: {score : 3}
```

问题解决了——这个时刻之后，任何人请求这条记录，都会取到最新的值。

2. 实践中的向量时钟

我们用 Riak 运行一遍前面例子所描述的场景。

在这个例子中，我们希望看到所有的冲突版本，以便能通过将它们手动解决。通过设置 `animals` 桶的 `allow_mult` 属性，可以保存多个版本的记录。任何包含多个值的键，称为同级数据（sibling value）。

```
$ curl -X PUT http://localhost:8091/riak/animals \
  -H "Content-Type: application/json" \
  -d '{"props":{"allow_mult":true}}'
```

这里，Bob 以 PUT 方法在系统中提交记录，其中包含评分 3 以及他的客户端 ID bob。

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
  -H "X-Riak-ClientId: bob" \
  -H "Content-Type: application/json" \
  -d '{"score" : 3}'
```

Jane 和 Rakshith 都取到 Bob 所创建的数据（只在这里显示了向量时钟；你得到的头信息会多很多。

注意，Riak 对 Bob 的 vclock 做了编码，但是在编码的背后，它依然包含客户端和版本（以及时间戳，所以你的查询结果会与这里看到的有所不同）信息。

```
$ curl -i http://localhost:8091/riak/animals/bruiser?return_body=true
X-Riak-Vclock: a85hYGBgzGDKBVIs7NtEXmUwJTLmsTI8FMs5zpcFAA==

{"score" : 3}
```

Jane 将评分更新为 2，并包含了她从 Bob 的版本中获取的最新向量时钟。对 Riak 而言，这是一个信号，告诉 Riak 她提交的值是对 Bob 的版本的更新。

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
```

```
  -H "X-Riak-ClientId: jane" \
  -H "X-Riak-Vclock: a85hYGBgzGDKBVIs7NtEXmUwJTLmsTI8FMs5zpcFAA==" \
  -H "Content-Type: application/json" \
  -d '{"score" : 2}'
```

由于 Jane 和 Rakshith 同时获取 Bob 的数据，因此 Rakshith 也基于 Bob 的向量时钟提交了一个更新（评分为 4）。

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
  -H "X-Riak-ClientId: rakshith" \
  -H "X-Riak-Vclock: a85hYGBgzGDKBVIs7NtEXmUwJTLmsTI8FMs5zpcFAA==" \
  -H "Content-Type: application/json" \
  -d '{"score" : 4}'
```

当 Jane 复查评分时，看到的不是预期的数据，而是代表多个选择的 HTTP 编码以及包含两个同级数据的 HTTP 响应体。

```
$ curl http://localhost:8091/riak/animals/bruiser?return_body=true
Siblings:
637aZSiky628lx1YrstzH5
7F85FBAIW8eiD9ubsBAeVk
```

Riak 以 multipart 格式存储这些版本，所以她能通过接受 MIME 类型，获取完整的对象。

```
$ curl -i http://localhost:8091/riak/animals/bruiser?return_body=true \
  -H "Accept: multipart/mixed"

HTTP/1.1 300 Multiple Choices
X-Riak-Vclock: a85hYGBgyWDKBVHs20Re...OYn9XY4sskQUA
Content-Type: multipart/mixed; boundary=1QwWn1ntX3gZmYQVBG6mAZRVXlu
Content-Length: 409

--1QwWn1ntX3gZmYQVBG6mAZRVXlu
Content-Type: application/json
Etag: 637aZSiky628lx1YrstzH5

{"score" : 4}
-1QwWn1ntX3gZmYQVBG6mAZRVXlu
Content-Type: application/json
Etag: 7F85FBAIW8eiD9ubsBAeVk
```

```
{"score" : 2}
--1QwWn1ntX3gZmYQVBG6mAZRVXlu–
```

请注意，前面所示的同级数据是设置为特定值的 HTTP Etag（Riak 称之为 vtag）。一个值得注意的有趣方面是，可以指定 URL 中的 vtag 参数，只检索对应的版本：`curl http://localhost:8091/riak/animals/bruiser?vtag=7F85FBAIW8eiD9ubsBAeVk` 会返回 `{"score" : 2}`。

Jane 现在的工作是用这点信息，做出合理的更新。她决定取两个评分的均值 3 分，用给定的向量时钟解决冲突。

```
$ curl -i -X PUT http://localhost:8091/riak/animals/bruiser?return_body=true \
-H "X-Riak-ClientId: jane" \
-H "X-Riak-Vclock: a85hYGBgyWDKBVHs20Re...OYn9XY4sskQUA" \
-H "Content-Type: application/json" \
-d '{"score" : 3}'
```

如果 Bob 和 Rakshith 现在检索 bruiser 的信息，他们会得到解决冲突后的评分。

```
$ curl -i http://localhost:8091/riak/animals/bruiser?return_body=true
HTTP/1.1 200 OK
X-Riak-Vclock: a85hYGBgyWDKBVHs20Re...CpQmAkonCcHFM4CAA==

{"score" : 3}
```

任何将来的请求都会返回 3 分。

3. 时间不断增长

你或许已经注意到向量时钟会随着越来越多的客户端更新操作，而不断增长。这是向量时钟的一个基本问题，对此 Riak 开发人员早已意识到了。他们扩展向量时钟，让它随着时间推移不断“修剪”，从而保持足较小的大小。Riak 修剪老的向量时钟的比率定义于桶的属性中，可以通过浏览桶，查看这个比率属性（以及桶的其他所有属性）。

```
$ curl http://localhost:8091/riak/animals
```

你会看到如下一些属性，它们规定了 Riak 如何在向量时钟变得太大之前，进行修剪。

```
"small_vclock":10,"big_vclock":50,"young_vclock":20,"old_vclock":86400
```

small_vclock 和 big_vclock 决定了向量的最小和最大长度，而 young_vclock 与 old_vclock 则描述了 vclock 在修剪之前的最小和最大年龄。

你可以在线阅读向量时钟与修剪的更多内容。[1]

4. 提交前/后钩子程序

Riak 可以在存储对象之前或者之后，通过钩子程序转化数据。提交前/后钩子程序是在提交数据前后执行的简单的 JavaScript（或者 Erlang）函数。提交前函数能以某种方式修改传入对象（甚至可以使提交失败），而提交后函数可以对成功的提交做出响应（例如，写入日志或者向某个目的地址发送电子邮件）。

每个节点服务器都有一个名为 `app.config` 的文件，需要在其中引用定制 JavaScript 程序的位置。首先，打开服务器 dev1 的配置文件，路径为 `dev/dev1/etc/app.config`，找到 `js_source_dir` 所在行。然后，将它替代为你期望的任何路径（注意，该行可能被%字符注释掉，所以得先删除%）。修改之后的行如下：

```
{js_source_dir,"～/riak/js_source"},
```

需要重复三次这种修改，每个开发服务器一次。不妨创建一个验证器，在提交之前解析传入的数据，保证输入中有评分信息并在 1～4 分之间。一旦违背这些标准，就会抛出错误，验证器会返回 JSON 对象，其中仅包含`{"fail" : message}`，这里的 `message` 可以是我们希望返回给用户的任何信息。如果数据是期望中的，只需要返回对象，Riak 会将它存储起来。

riak/my_validators.js
```
function good_score(object) {
  try {
  /* from the Riak object, pull data and parse it as JSON */
  var data = JSON.parse( object.values[0].data );
  /* if score is not found, fail here */
  if( !data.score || data.score === '' ) {
    throw( 'Score is required' );
  }
  /* if score is not within range, fail here */
  if( data.score < 1 || data.score > 4 ) {
    throw( 'Score must be from 1 to 4' );
  }
    catch( message ) {
```

[1] http://wiki.basho.com/Vector-Clocks.html

```
        /* Riak expects the following JSON if a failure occurs */
        return { "fail" : message };
      }
    /* No problems found, so continue */
    return object;
}
```

把这个文件存放在你所设置的文件夹 `js_source_dir` 中。由于改变了核心服务器的配置，因此需要用 `restart` 参数重启所有开发服务器。

```
$ dev/dev1/bin/riak restart
$ dev/dev2/bin/riak restart
$ dev/dev3/bin/riak restart
```

Riak 会扫描所有`.js` 扩展名的文件，并将它们加载到内存中。可以将桶的 `precommit` 属性设置为 JavaScript 的函数名（不是文件名），这样就能使用该 JavaScript 函数。

```
curl -X PUT http://localhost:8091/riak/animals \
-H "content-type:application/json" \
-d '{"props":{"precommit":[{"name" : "good_score"}]}}'
```

通过设置一个大于 4 的评分，来测试新的钩子程序。提交前钩子程序强制评分的范围在 1～4 之间，所以下面的操作将会失败：

```
curl -i -X PUT http://localhost:8091/riak/animals/bruiser \
-H "Content-Type: application/json" -d '{"score" : 5}'
HTTP/1.1 403 Forbidden
Content-Type: text/plain
Content-Length: 25

Score must be 1 to 4
```

你会得到禁止访问的代码 403，以及在“fail”域中返回的普通文本错误消息。如果用 GET 方法查询 `bruiser` 的值，它的评分依然是 3。尝试把评分设置为 2，操作就会成功。

提交后钩子程序与提交前类似，只是发生在提交成功之后。因为只能用 Erlang 编写提交后钩子程序，所以在此会略过这部分内容。事实上，也可以编写 Erlang 实现的 mapreduce 函数。但是 Riak 之旅将在其他预置模块与扩展部分继续。

3.4.2 扩展 Riak

Riak 的若干扩展在 Riak 交付的版本中是默认关闭的，这些扩展实现了一些新的行为，你会发现它们很有用。

1. 搜索 Riak

Riak 搜索会扫描 Riak 集群内的数据，并对此建立倒排索引（inverted index）。你或许会从第 2 章想起术语倒排索引（GIN index 代表 Generalized Inverted Index）。正如 GIN 一样，Riak 索引的存在，使各种字符串搜索在分布式环境中快速高效。

使用 Riak 搜索需要在 `app.config` 文件中启用它`，将 Riak 搜索配置设置为 `enabled，true`。

```
%% Riak Search Config
{riak_search, [
  %% To enable Search functionality set this 'true'.
  {enabled, true}
]},
```

如果你熟悉 Lucene 之类的搜索引擎，这部分应该是小菜一碟。如果不是，很容易掌握其要点。

需要通过提交前钩子程序，让 Riak 搜索引擎知道我们在数据库中改变了数据。可以在新的 `animals` 桶中安装 `riak_search_kv_hook`，它是 Erlang 模块的提交前函数，具体命令如下：

```
$ curl -X PUT http://localhost:8091/riak/animals \
-H "Content-Type: application/json" \
-d '{"props":{"precommit":
[{"mod": "riak_search_kv_hook","fun":"precommit"}]}}'
```

调用命令 `curl http://localhost:8091/riak/animals`，会显示钩子程序已经加到 `animals` 桶的 `precommit` 属性。现在，当向 `animals` 桶以 PUT 方法提交 JSON 或者 XML 编码的数据时，Riak 搜索引擎会对名字与值进行索引。我们上传一些新的小动物。

```
$ curl -X PUT http://localhost:8091/riak/animals/dragon \
-H "Content-Type: application/json" \
-d '{"nickname" : "Dragon", "breed" : "Briard", "score" : 1 }'
$ curl -X PUT http://localhost:8091/riak/animals/ace \
```

```
-H "Content-Type: application/json" \
-d '{"nickname" : "The Wonder Dog", "breed" : "German Shepherd", "score" : 3 }'
$ curl -X PUT http://localhost:8091/riak/animals/rtt \
-H "Content-Type: application/json" \
-d '{"nickname" : "Rin Tin Tin", "breed" : "German Shepherd", "score" : 4 }'
```

有几种选择可以查询这项数据，不妨使用 Riak 的 HTTP Solr 接口（实现了 Apache 的 Solr[1]搜索接口）。为搜索`/animals`，访问这样的 URL，`/solr` 后跟桶名`/animals`，以及`/select` 命令。而参数指定了搜索条件。

这里选择任何包含单词 Shepherd 的品种。

```
$ curl http://localhost:8091/solr/animals/select?q=breed:Shepherd
<?xml version="1.0" encoding="UTF-8"?>
<response>
  <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">1</int>
    <lst name="params">
      <str name="indent">on</str>
      <str name="start">0</str>
      <str name="q">breed:Shepherd</str>
      <str name="q.op">or</str>
      <str name="df">value</str>
      <str name="wt">standard</str>
      <str name="version">1.1</str>
      <str name="rows">2</str>
    </lst>
  </lst>
  <result name="response" numFound="2" start="0" maxScore="0.500000">
    <doc>
      <str name="id">ace</str>
      <str name="breed">German Shepherd</str>
      <str name="nickname">The Wonder Dog</str>
      <str name="score">3</str>
    </doc>
    <doc>
      <str name="id">rtt</str>
      <str name="breed">German Shepherd</str>
      <str name="nickname">Rin Tin Tin</str>
```

[1] http://lucene.apache.org/solr/

```
      <str name="score">4</str>
    </doc>
  </result>
</response>
```

如果你喜欢该查询返回 JSON，不妨增加参数 `wt=json`。可以在查询中合并多个参数，只需用空格（或者 URL 编码形式的%20）隔开它们，并将参数 `q.op` 设置为 `and`。要找到 nickname 中包含 rin，且 breed 中包含 shepherd 的品种，执行下面的命令：

```
$ curl http://localhost:8091/solr/animals/select\
?wt=json&q=nickname:rin%20breed:shepherd&q.op=and
```

Riak 搜索允许更多丰富多彩的查询语法，比如，通配符（用*匹配多个字符，用?匹配一个字符），尽管只能用在搜索项的末尾。查询 `nickname:Drag*`会匹配 Dragon，但是 `nickname:*ragon` 不会匹配任何数据。

范围搜索也是很好的一种选择。

```
nickname:[dog TO drag]
```

基于布尔运算符、分组与邻近搜索，可以实现更多的复杂查询。除此之外，可以指定自定义的数据编码，建立自定义索引，甚至在搜索时选择两者之一。也可以在下表中找到其他 URL 参数：

参　数	描　述	默 认 值
Q	给定的查询字符串	
q.op	查询项之间是 `and` 还是 `or`	`or`
Sort	排序依据的字段名	none
Start	匹配列表中返回的第一个对象	0
Rows	返回结果的最大行数	20
Wt	输出是 `xml` 还是 `json`	`xml`
Index	指定要用的索引	

关于 Riak 的搜索扩展，还有大量内容可以学习，远远多于这里所能讨论的内容。理想情况下，你已能感受到 Riak 搜索的强大。如果你打算为大规模 Web 应用提供搜索功能，Riak 是一个明确的选择，但是你若需要大量简单的自由定义的查询，有必要再次评估你的选择。

2. 索引 Riak

在 1.0 版本，Riak 支持二级索引。这与我们在 PostgreSQL 中看到的索引类似，但是也略有不同。Riak 允许你对附在对象头的元数据索引，而不是对特定的列或者数据列索引。

再次说明，必须修改 app.config 文件。将存储后端从 bitcask 切换到 eLevelDB，然后重启服务器，如下所示：

```
{riak_kv, [
  %% Storage_backend specifies the Erlang module defining the
  %% storage mechanism that will be used on this node.
{storage_backend, riak_kv_eleveldb_backend},
```

有种 Google 键值存储称为 LevelD[1]，它的 Erlang 实现就是 eLevelDB。这个新的后端实现允许在 Riak 中出现二级索引。

当系统一切就绪准备运行时，可以对任意对象索引，这些对象带有任意数量称为索引条目的头标签，它们定义了对象以何种方式索引。标签名以 x-riak-index-开始，以_int 或者_bin 结尾，分别表示整数值或者二进制值（整数外的任何数据）。

为增加昵称为 Blue II 的小狗，一只巴特勒牛头犬吉祥物（Bulter Bulldogs mascot），我们将按照大学名（事实上，这只小狗是巴特勒大学的吉祥物）以及版本号（Blue 2 是第二只牛头犬吉祥物）索引。

```
$ curl -X PUT http://localhost:8098/riak/animals/blue
-H "x-riak-index-mascot_bin: butler"
-H "x-riak-index-version_int: 2"
-d '{"nickname" : "Blue II", "breed" : "English Bulldog"}'
```

或许你已经注意到，索引与存储在键中的值没有什么关系。这实际上是一个强大的功能，这种特性让我们可以索引数据，同时对所存储的任意数据保持正交。如果你想把一个视频另存为值，你仍然可以对它索引。

通过索引获取值是相当简单直接的。

```
$ curl http://localhost:8098/riak/animals/index/mascot_bin/butler
```

尽管 Riak 中的二级索引是往正确方向迈出的一大步，但是前方的路依然很长。例如，如果你想对日期索引，你必须存储一个能排序的字符串，比如“YYYYMMDD”。而存储浮点数要求首先以 10 的某个有效精度的倍数，处理浮点；然后再将它另存为整数——比如，

[1] http://code.google.com/p/leveldb/

1.45*100==145。而这种转化是由客户端负责的。然而，在 mapreduce、Riak 搜索以及二级索引之间，Riak 还是提供了许多工具，另辟蹊径，以超越简单键的值访问，放宽了键值存储设计中的经典约束。

3.4.3 第 3 天总结

伴随着一些更进阶的概念，我们完成了 Riak 的学习：如何通过向量时钟处理版本冲突，如何使用提交钩子程序（commit hook）验证或修改输入数据。我们还深入讨论了使用一组 Riak 扩展：激活 Riak 搜索，以及对数据索引，使用户获得更多的查询灵活性。将这些概念，连同第 2 天的映射归约（mapreduce）和第 1 天的链接一起使用，你能创建一个灵活的工具组合，远远超出标准的键值存储。

第 3 天作业

求索

1. 找到 Riak 函数贡献列表库（提示：在 GitHub 里）。
2. 阅读更多关于向量时钟的内容。
3. 学习创建你自己的索引配置。

实践

1. 创建你自己的索引，定义 `animals` 的模式。具体而言，把 `score` 域设置为整数类型，按范围查询。
2. 启动一个小集群，包含三个服务器（例如，三个笔记本电脑或者 EC2[1]虚拟机实例），并分别安装 Riak。将这些服务器设置为一个集群。安装 Google 股票数据集，可以在 Basho 网站上找到[2]。

3.5 总结

Riak 是我们介绍的第一个非 SQL 型数据库。它是分布式的、有数据副本的、键值增强的存储方式，且不会发生单点故障。

如果你截至目前的数据库经验都限于关系数据库，Riak 也许看起来像一个陌生的野

[1] http://ec2.amazonaws.com/
[2] http://wiki.basho.com/Loading-Data-and-Running-MapReduce-Queries.html

兽。事务、SQL 与模式没有了。有的是键（key），而桶（bucket）之间的链接一点不像数据表的联接，并且 mapreduce 又是一种很难上手的方法论。然而，凡事都有利弊，Riak 很适合解决某类问题。它可以扩展更多的节点服务器（而非扩展规模更大的单个服务器），而其易于使用的特点，是解决 Web 独特可扩展性问题的一次极佳尝试。基于 HTTP 结构之上的 Riak，并非是“炒冷饭”，而是在最大程度上，为各种框架或者 Web 系统增加了灵活性。

3.5.1　Riak 的优点

如果你想设计一个类似 Amazon 的大型订货系统，或者你最关注的是高可用性，这时，你应该考虑一下 Riak。无疑，Riak 的优势之一就是它致力于避免单点故障，设法支持最大的正常运行时间，并且增加（或者缩小）规模以适应变化的需求。如果你没有复杂的数据，Riak 让一切维持简单的状态，然而一旦你需要，它也可以处理相当复杂的数据。目前，Riak 支持许多编程语言（你能在 Riak 网站上找到相关内容），但是如果你使用 Erlang，就能获得扩展 Riak 核心的能力。你若需要超出 HTTP 处理能力的速度，也可以试试通过 Protobuf[17] 进行通信，它是一种更有效率的二进制编码和传输协议。

3.5.2　Riak 的缺点

如果你需要简单的查询能力、复杂的数据结构或者严格的模式，又或者你不需要节点服务器进行横向扩展，那么 Riak 可能不是你的最佳选择。我们对于 Riak 的主要抱怨之一是，作为自由定义（ad hoc）的查询框架，它依然不够简单而健壮，尽管 Riak 确实在正确的轨道上前进。mapreduce 提供了无与伦比的强大功能，但是我们希望有更多内置的基于 URL 或者其他 PUT 查询的操作。索引的加入是在这个方向上的一大进步，也是我们乐意详述的一个概念。最后，如果你不想编写 Erlang 代码，你会发现使用 JavaScript 的一些限制，比如，不能实现提交后处理，或者，较慢的 mapreduce 执行过程。但是，Riak 团队正在解决这些相对较小的问题。

3.5.3　Riak 之于 CAP

Riak 提供了一个聪明的方法，以规避 CAP 施加于所有分布式数据库上的约束。相比像 PostgreSQL 这样只支持强写入一致性的系统，Riak 对于这个问题的处理能力是惊人的。它利用了 Amazon Dynamo 论文的深刻观点——CAP 可以在每个桶或者每个请求的基础上改变。它的健壮与灵活是开源数据库系统发展的一大进步。当你读到本书中的其他数据库时，想起 Riak，你依然会惊叹于它的灵活性。

3.5.4 结束语

如果你需要存储海量数据目录，采用其他方式很可能没有 Riak 做得好。尽管 40 多年来，人们没有在这方面对关系数据库进行研究和调整，但并非所有问题都需要遵循 ACID 或者强制使用数据模式。如果你想把数据库嵌入一个设备或者处理金融事务，你该避免使用 Riak。如果你想获得扩展的能力或者在 Web 上提供数据，请考虑使用 Riak。

第 4 章

HBase

Apache 的 HBase 是用来做大事的，就像射钉枪。你不应该用 HBase 来记录公司的销售清单，就像你不会用射钉枪来造玩具屋一样。如果你的数据不是用多少 GB 来衡量，那么你可能需要一个小点的工具。

HBase 初看起来很像关系数据库，如果你不知道更多的信息，你可能认为它就是一个关系数据库。当学习 HBase 时，最有挑战性的任务不是技术，而是 HBase 中用到的许多词语，我们似乎很熟悉，但却具有欺骗性，并不是原来的含义。例如，HBase 将数据存放在桶（bucket）中，它称为表，其中包含单元（cell），单元是行和列的交叉处。到目前为止很好，是吗？

错了！在 HBase 中，表的行为不像关系，行不像记录，列是完全可变的（没有受到模式描述的强制限制）。模式的设计仍然重要，因为它告知了系统的性能特点，但它不会替你保持房间的整洁。HBase 是 RDBMS 的邪恶孪生兄弟，逆反超人。

那么，你为什么要用这个数据库？除了伸缩性之外，还有一些其他理由。首先，HBase 有一些内置的特性，其他数据库没有，诸如版本管理、压缩、垃圾回收（对于超期的数据），以及内存表。直接提供这些特性意味着，如果你用到它们，就可以少写代码。HBase 也提供了很强的一致性保证，容易实现从关系数据库的迁移。

由于这些原因，HBase 很出色地成为在线分析处理系统的基石。虽然与其他数据库相比，单个操作可能要慢一些，但 Hbase 擅长做的事情是扫描巨大的数据集。所以，对于真正巨大的查询，HBase 通常胜过其他数据库。这也解释了为什么 HBase 常用于大公司，作为后台日志和查询系统。

4.1 介绍 HBase

HBase 是面向列的数据库，在一致性和伸缩性方面引以为傲。它基于 BigTable 的思想。BigTable 是 Google 开发的高性能专有数据库，在 2006 年的白皮书“Bigtable: A Distributed Storage System for Structured Data”中介绍过[1]。HBase 最初是为处理自然语言而创建的，开始是作为 Apache Hadoop 的一个贡献包。从此之后，它成为一个顶级 Apache 项目。

在架构方面，HBase 的设计初衷是能够容错。对于单个机器，硬件故障可能不常见，但对于大型集群，节点故障是常态。通过预写式日志（write-ahead logging）和分布式配置，HBase 能够快速地从单个服务器故障中恢复。

另外，HBase 所处的生态系统也带来了额外的好处。HBase 基于 Hadoop，Hadoop 是一个坚固的、可伸缩的计算平台，提供了分布式文件系统和 mapreduce 的计算能力。在有 HBase 的地方，就有 Hadoop 和其他基础设施组件，你可以在自己的应用中加以利用。

一些知名度很高的公司在积极地使用和开发 HBase，以解决它们的“大数据”问题。例如，Facebook 在 2010 年 11 月宣布了新的消息基础设施，选择了 HBase 作为其主要组件。Stumbleupon 几年以来一直采用 Hbase 来实现实时数据存储和分析，直接通过 HBase 提供各种网站特征。Twitter 大量采用 HBase，从数据生成（针对找人这样的应用）到保存监控/性能数据。使用 HBase 的公司还包括 eBay、Meetup、Ning、Yahoo 等。

在这些支持下，HBase 的新版本发布的速度很快。在编写本书时，当前的稳定版是 0.90.3，也就是我们采用的版本。现在下载 HBase，我们就要开始了。

4.2 第 1 天：CRUD 和表管理

今天的目标是学习使用 HBase 的基本特性。我们会让一个 HBase 本地实例以单机模式运行，然后通过 HBase shell 来创建和改变表，利用基本命令来插入、修改数据。之后，我们将探讨如何在 JRuby 中利用 HBase 的 Java API，通过编程的方式来执行其中一些操作。在这个过程中，我们将介绍 HBase 的一些架构概念，如表中的行、列族（column family）、列和值之间的关系等。

完全可运营的、产品级的 HBase 集群，实际上至少应该有 5 个节点，常规经验是这样的。这样的配置对我们的需求来说是杀鸡用牛刀了。幸运的是，HBase 支持 3 种运行模式：

- 单机模式是单台机器独立运行；

[1] http://labs.google.com/papers/bigtable.html

- 伪分布式模式是单个节点伪装一个集群；
- 完全分布式模式是一群节点一起工作。

在本章的大部分时间，我们采用单机模式运行 HBase。即使这样也有一点挑战性，所以我们虽然不会全面介绍安装和管理，但会在合适的时候，给出一些相关的故障诊断提示。

4.2.1　配置 HBase

在使用 HBase 之前，必须先配置它。HBase 的配置设置保存在一个文件中，名为 hbase-site.xml，在${HBASE_HOME}/conf/目录下。请注意，HBASE_HOME 是一个环境变量，指向 HBase 的安装目录。

初始情况下，这个文件只包含一个空的<configuration>标签。可以采用下面的格式，在配置文件中添加任意多个属性定义：

```
<property>
  <name>some.property.name</name>
  <value>A property value</value>
</property>
```

所有可用属性的完整清单，以及默认值及其描述信息，都在文件 hbase-default.xml 中，位于${HBASE_ HOME}/src/main/resources 目录下。

默认情况下，HBase 用一个临时目录来存放它的数据文件。这意味着每当操作系统决定回收这些磁盘空间时，你的所有数据都会丢失。

要保存数据，应该指定一个稳定的存储位置。可以设置 hbase.rootdir 属性，指向一个合适的路径，像这样：

```
<property>
  <name>hbase.rootdir</name>
  <value>file:///path/to/hbase</value>
</property>
```

要启动 HBase，打开一个终端窗口（命令行提示），并执行下面的命令：

```
${HBASE_HOME}/bin/start-hbase.sh
```

要关闭 HBase，使用同一目录下的 stop-hbase.sh 命令。

如果出了状况，请在${HBASE_ HOME}/logs 目录下，查看最近修改的文件。在*nix 系统中，下面的命令将最近的日志数据原封不动地输出到控制台：

```
cd ${HBASE_HOME}
find ./logs -name "hbase-*.log" -exec tail -f {} \;
```

4.2.2 HBase shell

HBase shell 是基于 JRuby 的命令行程序，可以用它与 HBase 交互。在 shell 中，可以添加和删除表，改变表的模式，添加或删除数据，并完成一些其他任务。后面会探讨连接 HBase 的其他方式，但现在用 shell 就够了。当 HBase 运行时，打开一个终端，并启动 HBase 的 shell：

```
${HBASE_HOME}/bin/hbase shell
```

要确认它工作正常，请查询它的版本信息。

```
hbase> version
0.90.3, r1100350, Sat May 7 13:31:12 PDT 2011
```

任何时候都可以输入 `help`，查看可用命令的清单，或某个命令的用法。

接下来，执行 `status` 命令，查看 HBase 服务器的状态。

```
hbase> status
1 servers, 0 dead, 2.0000 average load
```

如果这些命令出现任何问题，或 shell 挂起，可能是连接出了问题。HBase 尽了最大努力，根据网络设置来自动配置它的服务，但有时候它也会出错。如果看到这些症状，请查看 4.2.3 小节中的“HBase 的网络设置”。

4.2.3 创建表

映射表是一个键值对，就像 Rubu 中的哈希，或 Java 中的 hashmap。HBase 中的表基本上是一个很大的映射表，更准确地说，它是嵌套许多映射表的映射表。

> **HBase 的网络设置**
>
> 默认情况下，HBase 试图向外部客户提供它的服务，但在例子中，只需要在同一台机器上发起连接。所以，在 `hbase-site.xml` 中加入下列部分或全部属性，也许会有帮助（你的用处可能不一样）。请注意，仅当你打算进行本地连接，而非远程连接时，下面表中的这些值才有帮助。

属　性　名	值
hbase.master.dns.interface	lo
hbase.master.info.bindAddress	127.0.0.1
hbase.regionserver.info.bindAddress	127.0.0.1
hbase.regionserver.dns.interface	lo
hbase.zookeeper.dns.interface	lo

这些属性告诉 HBase，如何为主服务器和区域服务器（稍后将讨论它们）建立连接，以及 zookeeper 配置服务。设置为“lo”的属性指的是所谓的 loopback（回送）网卡。在*nix 系统中，回送网卡不是真实的网卡（与以太网或无线网卡不同），而是只有软件模拟的网卡，让计算机连接它自己。`bindAddress` 属性告诉 HBase，它尝试监听哪个 IP 地址。

在 HBase 表中，键可以是任意字符串，每个键都映射到一行数据。行本身也是一个映射表，其中的键称为列，而值就是未解释的字节数组。列按照列族（column family）进行分组，所以列的完全名称包含两个部分：列族名称和列限定符（column qualifier）。通常它们用一个冒号连接起来（例如，'family:qualifier'）。

为了展示这些概念，请看图 4-1。

图 4-1　HBase 的表包含行、键、列族、列和值

在这张图中，有一张假想的表，它有两个列族：`color` 和 `shape`。该表有两行（虚线框表示），由行键来唯一标识：`first` 和 `second`。请看第一行，我们看到它在列族 `color` 中有 3 列（限定符分别是 `red`、`blue` 和 `yellow`），在列族 `shape` 中有 1 列（`square`）。行键和列名（包括列族和限定符）的组合，形成了定位数据的地址。在这个例子中，三元组 `first/color:red` 指向了值 `'#F00'`。

现在，让我们利用所学的表结构知识，做点有趣的事情：我们要做一个 wiki！

我们可能希望为 wiki 加上很多生动的信息，但我们会先从最基本的部分开始。一个 wiki 包含了一些页面，每个页面都有一个唯一的标题字符串，并包含一些文本内容。

执行 create 命令来创建 wiki 表：

```
hbase> create 'wiki', 'text'
0 row(s) in 1.2160 seconds
```

这里创建了一个名为 wiki 的表，只有一个名为 text 的列族。表目前是空的，它没有行，因此也没有列。不像关系数据库，HBase 的列是针对它所属的行的。如果添加行，同时就会添加列来保存数据。

将表体系结构画出来，就得到了图 4-2。按惯例，我们预期每行在 text 列族中只有一列，限定符是空字符串（"）。所以，包含一页文本的完整列名是'text:'。

当然，wiki 表要有用，就需要有内容。让我们来添加一些数据！

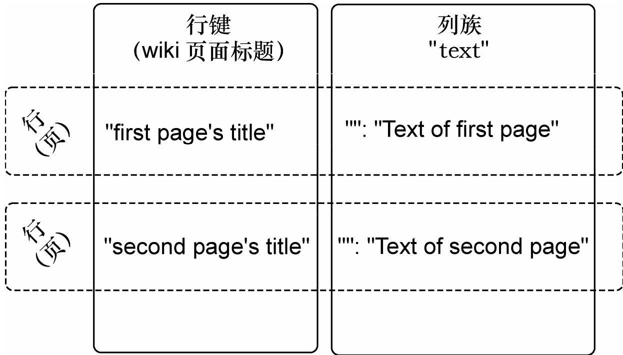


图 4-2　wiki 表有一个列族

4.2.4　插入、更新和读取数据

wiki 需要有一个首页，所以从这里开始。在 HBase 的表中添加数据，使用 put 命令：

```
hbase> put 'wiki', 'Home', 'text:', 'Welcome to the wiki!'
```

这个命令在 wiki 表中插入一行，键是'Home'，将'Welcome to the wiki!'加入到'text:'列中。

可以用 get 来查询'Home'行的数据，它需要两个参数：表名和行键。可以选择性地指定要返回哪些列。

```
hbase> get 'wiki', 'Home', 'text:'
COLUMN CELL
text: timestamp=1295774833226, value=Welcome to the wiki!
1 row(s) in 0.0590 seconds
```

请注意输出中的 `timestamp`（时间戳）字段。HBase 为所有数据值保存一个整数时间戳，表示自某个时间（1970 年 1 月 1 日 00:00:00 UTC）以来的毫秒数。如果把新的值写入同一个单元，老的值将留在附近，按它的时间戳索引。这是非常出色的特征。大部分数据库要求你自己专门处理历史数据，但在 HBase 中，版本管理已经内置支持了！

Put 和 Get

put 和 get 命令可以明确指定时间戳。如果你不想使用自某个时间以来的毫秒数，也可以指定另一个选择的整数值。这让你在需要时可以在时间维上工作。如果你不指定时间戳，HBase 会在插入时使用当前的时间，在读取时返回最近的版本。

如果要看一个公司如何选择覆写时间戳字段的例子，请参见 4.2.5 节的案例研究"Facebook 的消息索引表"。本章剩下的部分将采用默认的时间解释。

4.2.5 修改表

到目前为止，wiki 的模式包含的页面带有标题和文本，以及整数的版本历史，此外没有其他东西。让我们进一步扩展需求，包含以下内容：

- 在 wiki 中，页面由标题唯一确定；
- 页面可以有无数个新版本；
- 新版本由它的时间戳确定；
- 新版本包含文本，以及可选的提交说明；
- 新版本由一个作者提交，作者由名字唯一确定。

我们的需求可以画成草图，如图 4-3 所示。在这个页面需求的抽象表示中，我们看到每个新版本都有一个作者，一段提交说明，一些内容文本和一个时间戳。页面的标题不是新版本的一部分，因为它是标识符，所以用它来说明这些新版本属于同一个页面。

将理念映射到 HBase 表，需要采取有点不同的方式，如图 4-4 所示。wiki 表使用标题作为行键，将其他页面数据分在两个列族中，分别是 `test` 和 `revision`。列族 `text` 和以前一样，我们希望每行都只有一列，限定符是空串（''），来保存文章的内容。列族

`revision` 的任务是保存新版本特有的其他数据，如作者和提交说明。

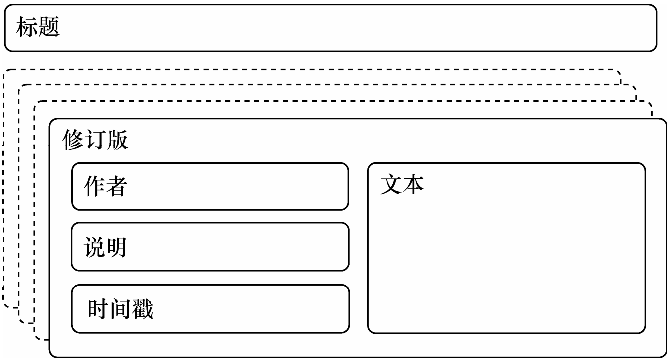


图 4-3 wiki 页面的需求（包含时间维度）

	行键（标题）	列族 "text"	列族 "revision"
行（页）	"first page"	"": "..."	"author": "..." "comment": "..."
行（页）	"second page"	"": "..."	"author": "..." "comment": "..."

图 4-4 更新的 wiki 表结构（时间维度没显示）

案例研究：Facebook 的消息索引表

Facebook 采用 HBase 作为其消息基础设施的主要组件，用于存放消息数据，也为查询而维护倒排索引（inverted index）。

在它的索引表模式中：

- 行键是用户 ID；
- 列限定符是出现在用户消息中的单词；
- 时间戳是包含这个单词的消息的 ID。

因为用户之间的消息是不可修改的，所以消息的索引条目也是静态的。版本管理的概念没有意义。

对于 Facebook，让时间戳匹配消息 ID，这让它们有了另一个维度来存储数据。

1. 默认值

因为在创建 wiki 表时没有带特殊选项，所以全部采用了 HBase 的默认值。因为其中有一个默认值，即列数据只保存 3 个版本，所以调高这个值。更新的 wiki 表结构如图 4-4 所示。如果要改动模式，必须先用 disable 命令，让表离线。

```
hbase> disable 'wiki'
0 row(s) in 1.0930 seconds
```

现在可以用 alter 命令，修改列族的属性。

```
hbase> alter 'wiki', { NAME => 'text', VERSIONS =>
hbase* org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }
0 row(s) in 0.0430 seconds
```

这里，我们告诉 HBase 修改 text 列族的 VERSIONS 属性。还有一些其他属性可以设置，第 2 天将讨论其中的一部分。hbase*这一行表示，这是前一行的续行。

2. 修改表

修改列族属性的操作可能开销很大，因为 HBase 必须用选定的规格创建新的列族，然后复制所有的数据。在生产系统中，这可能导致相当长的停机时间。出于这个原因，事先设置列族属性是较好的做法。

在 wiki 表仍然禁用的情况下，添加 revision 列族，还是使用 alter 命令：

```
hbase> alter 'wiki', { NAME => 'revision', VERSIONS =>
hbase* org.apache.hadoop.hbase.HConstants::ALL_VERSIONS }
0 row(s) in 0.0660 seconds
```

像之前的 text 列族一样，只是在表模式中添加了一个 revision 列族，而不是个别列。虽然我们希望每行最终都包含 revision:author 和 revision:comment，但这依赖于客户来实现这个希望，它没有写进正式的模式中。如果有人想在页面中加入 revision:foo，HBase 也不会阻止这种做法。

3. 继续

添加列族之后，重新启用 wiki：

```
hbase> enable 'wiki'
0 row(s) in 0.0550 seconds
```

现在 wiki 表已经修改好了，可以支持扩展的需求清单，可以开始向 revision 列族的列

中添加数据。

4.2.6 通过编程方式添加数据

我们看到，HBase 的 shell 对于操纵表这样的任务是很好用的。遗憾的是，shell 对数据插入的支持不是最好的。put 命令一次只允许设置一个列值，而在刚才更新过的 shcema 中，需要同时添加多个列值，这样它们就有相同的时间戳。我们需要开始编写脚本。

下面的脚本可以直接在 HBase 的 shell 中执行，因为 shell 也是一个 JRuby 解释器。在执行时，它为主（Home）页添加新版本的文本（text），同时设置了作者（author）和说明（comment）字段。JRuby 运行在 Java 虚拟机（JVM）上，所以它能访问 HBase 的 Java 代码。这些例子在非 JVM 的 Ruby 上是不能执行的。

```
hbase/put_multiple_columns.rb
import 'org.apache.hadoop.hbase.client.HTable'
import 'org.apache.hadoop.hbase.client.Put'

def jbytes( *args )
  args.map { |arg| arg.to_s.to_java_bytes }
end

table = HTable.new( @hbase.configuration, "wiki" )

p = Put.new( *jbytes( "Home" ) )

p.add( *jbytes( "text", "", "Hello world" ) )
p.add( *jbytes( "revision", "author", "jimbo" ) )
p.add( *jbytes( "revision", "comment", "my first edit" ) )
table.put( p )
```

以 `import` 开头的两行将用到的 HBase 类引入到 shell。这让我们以后不用写出完整的名称空间。接下来，`jbytes()` 函数接受任意数量的参数，返回一个转换好的 Java 字节数组，这是 HBase 的 API 方法要求的。

然后，创建了一个本地变量（`table`），指向 wiki 表，使用`@hbase` 管理对象的配置信息。

然后，我们准备提交操作，即创建并准备一个新的 `Put` 实例，它记下要修改的行。在这个例子中，我们的关注点一直在处理的主页。最后，利用 `add()` 向 Put 实例添加属性，然后调用 `table` 对象来执行准备好的 `put` 操作。方法 `add()` 有几种形式，在例子中，使用了 3 个参数的版本：`add`（列族，列限定符，值）。

为什么要多个列族

在构造整个结构时，你可能不想用多个列族，为什么不把一行的所有数据都放在一个列族中呢？这个解决方案实现起来更简单。但不使用多个列族也有不好的一面，即丧失了

细粒度性能调优的机会。每个列族的性能选项是独立配置的。这些设置影响到读和写的速度，以及磁盘空间的占用。

HBase 中的所有操作在行级是原子的。对于读或写的行，不论影响到多少列，该操作都会有一致的视图。这种设计决定有助于客户对数据做出聪明的推断。

因为 put 操作影响到多个列，并且没有指定时间戳，所以所有的列值都会有同样的时间戳（以毫秒表示的当前时间）。让我们调用 get 来验证一下。

```
hbase> get 'wiki', 'Home'
COLUMN            CELL
revision:author   timestamp=1296462042029, value=jimbo
revision:comment  timestamp=1296462042029, value=my first edit
text:             timestamp=1296462042029, value=Hello world
3 row(s) in 0.0300 seconds
```

你可以看到，上面列出的每个列值都有相同的时间戳。

4.2.7 第 1 天总结

今天，我们亲自查看了运行的 HBase 服务器。我们学习了如何配置它，以及查看日志文件来诊断故障。利用 HBase shell，我们完成了基本管理和数据操作任务。

在对 wiki 系统的建模中，我们探索了 HBase 中的模式设计。我们学习了如何创建表和操作列族。设计 HBase 的模式意味着决定列族的一些选项，同样重要的是，我们对行键和时间戳这样的特性的语义解释。

我们也开始接触 HBase Java API，在 shell 中执行 JRuby 代码。在第 2 天，我们将更进一步，利用 shell 来执行定制的脚本，完成数据导入这样的大任务。

你应该已经开始摆脱关系数据库的一些概念负担了，如表、行和列。HBase 使用这些术语的方式与它们在其他系统中的含义有差别，当我们深入研究 HBase 的特性时，这种差别会更突出。

第 1 天作业

HBase 在线文档一般有两种风格：极其技术化和不存在。这种情况正在改变，“getting started”（初学指南）开始出现了，但要准备好花些时间，在 Javadoc 或源代码中寻找答案。

求索

1．弄清楚如何通过 shell 完成下面的工作：

- 删除一行中个别列的值；
- 删除一整行。

2．在书签中加入你用的 HBase 版本的 API 文档。

实践

1．创建一个函数 put_many()，它创建一个 Put 实例，在其中添加一些列-值对，并提交给一个表。函数的签名应该像这样：

```
def put_many( table_name, row, column_values )
  # 这里是你的代码
End
```

2．将 put_many()函数粘贴到 HBase 的 shell 中，即定义它，并像这样调用它：

```
hbase> put_many 'wiki', 'Some title', {
hbase*    "text:" => "Some article text",
hbase*    "revision:author" => "jschmoe",
hbase*    "revision:comment" => "no comment" }
```

4.3　第 2 天：处理大数据

第 1 天创建了表并进行了操作，现在要开始向 wiki 表添加一些正式的数据。今天，我们将针对 HBase API 编写脚本，最终将维基百科（Wikipedia）的内容加入到 wiki 中！在这个过程中，我们将利用一些性能方面的技巧，让导入工作更快完成。最后，我们将研究 HBase 的内部构造，看看它如何将数据分区，从而实现性能提升和灾难恢复。

4.3.1　导入数据，调用脚本

在试用新的数据库系统时，人们常常会遇到一个问题，即如何迁移数据。像第 1 天所做的那样，用静态字符串手动完成 Put 操作，这是不错的方法，但我们可以做得更好。

幸运的是，将命令粘贴到 shell 中并不是执行命令的唯一方法。在从命令行启动 HBase shell 时，可以指定要执行的 JRuby 脚本名。HBase 将执行该脚本，就像在 shell 中直接输入一样。语法是这样的：

```
${HBASE_HOME}/bin/hbase shell <your_script> [<optional_arguments> ...]
```

既然我们对“大数据”特别感兴趣，那就让我们创建脚本，将维基百科的文章导入

wiki 表。WikiMedia 基金会监管 Wikipedia、Wictionary 和其他项目，并定期发布数据转储，我们可以使用这些数据转储。这些数据转储是巨大的 XML 文件。这里是英文维基百科的示例记录：

```
<page>
  <title>Anarchism</title>
  <id>12</id>
  <revision>
    <id>408067712</id>
    <timestamp>2011-01-15T19:28:25Z</timestamp>
    <contributor>
      <username>RepublicanJacobite</username>
      <id>5223685</id>
    </contributor>
    <comment>Undid revision 408057615 by [[Special:Contributions...</comment>
  <text xml:space="preserve">{{Redirect|Anarchist|the fictional character|
...
[[bat-smg:Anarkėzmos]]
[[zh:□□□□□]]</text>
  </revision>
</page>
```

因为我们很明智，所以这里包含的所有信息都已体现在模式中：标题（行键）、文本、时间戳和作者。因此，我们应该能够写一个脚本来导入各个版本的数据，这不是很麻烦。

4.3.2 流式 XML

要事优先。需要以流的（SAX）方式，解析巨大的 XML 文件，我们就从这里开始。在 JRuby 中逐条解析 XML 文件的工作，看起来基本上是这样的：

```
hbase/basic_xml_parsing.rb
import 'javax.xml.stream.XMLStreamConstants'

factory = javax.xml.stream.XMLInputFactory.newInstance
reader = factory.createXMLStreamReader(java.lang.System.in)

while reader.has_next

  type = reader.next

  if type == XMLStreamConstants::START_ELEMENT
    tag = reader.local_name
    # do something with tag
  elsif type == XMLStreamConstants::CHARACTERS
    text = reader.text
```

```
    # do something with text
  elsif type == XMLStreamConstants::END_ELEMENT
    # same as START_ELEMENT
  end

end
```

分解一下，有几点值得一提。首先，生成了一个 XMLStreamReader，并将它与 java.lang. System.in 联系在一起，这意味着它将从标准输入读取数据。

接下来，设置了一个 while 循环，它不断从 XML 流中取出符号，直到取完为止。在这个 while 循环内部，处理当前的符号。该符号可以是 XML 的开始标签、结束标签，或标签之间的文本，这决定了要做的事。

4.3.3　流式维基百科

现在，我们可以将这个基本的 XML 处理框架，和前面探讨的 HTable 和 Put 接口结合起来。下面是得到的脚本。大部分代码应该看起来很熟悉，我们将讨论几个新增部分。

```
hbase/import_from_wikipedia.rb
require 'time'

import 'org.apache.hadoop.hbase.client.HTable'
import 'org.apache.hadoop.hbase.client.Put'
import 'javax.xml.stream.XMLStreamConstants'

def jbytes( *args )
  args.map { |arg| arg.to_s.to_java_bytes }
end

factory = javax.xml.stream.XMLInputFactory.newInstance
reader = factory.createXMLStreamReader(java.lang.System.in)

① document = nil
buffer = nil
count = 0

table = HTable.new( @hbase.configuration, 'wiki' )
② table.setAutoFlush( false )

while reader.has_next

  type = reader.next

③ if type == XMLStreamConstants::START_ELEMENT
```

```
      case reader.local_name
      when 'page' then document = {}
      when /title|timestamp|username|comment|text/ then buffer = []
      end

  ④ elsif type == XMLStreamConstants::CHARACTERS

      buffer << reader.text unless buffer.nil?

  ⑤ elsif type == XMLStreamConstants::END_ELEMENT

      case reader.local_name
      when /title|timestamp|username|comment|text/
        document[reader.local_name] = buffer.join
      when 'revision'

        key = document['title'].to_java_bytes
        ts = ( Time.parse document['timestamp'] ).to_i

        p = Put.new( key, ts )

        p.add( *jbytes( "text", "", document['text'] ) )
        p.add( *jbytes( "revision", "author", document['username'] ) )
        p.add( *jbytes( "revision", "comment", document['comment'] ) )

        table.put( p )

        count += 1
        table.flushCommits() if count % 10 == 0
        if count % 500 == 0
          puts "#{count} records inserted (#{document['title']})"
        end

      end

    end

  end

  table.flushCommits()

  exit
```

① 要指出的第 1 处不同是引入了几个变量。

- document：存放当前文章和新版本数据。

- buffer：存放文档中当前字段的字符数据（text、title、author 等）。
- count：追踪我们已经导入了多少文章。

② 特别要注意 table.setAutoFlush（false）的使用。在 HBase 中，数据自动定期刷新到磁盘。这对于大多数应用是适用的。该脚本禁用了 autoflush，所有 put 操作都会在缓存中执行，直到我们调用 table.flushCommits()。这让我们能够在方便的时候，批量地写入和执行。

③ 接下来，我们看看解析时发生了什么。如果开始标签是<page>，那么将 document 重置成一个空的哈希表。否则，如果我们关注的是其他标签，重置 buffer 来存放它的文本。

④ 处理字符数据的方法，是将它追加到 buffer 中。

⑤ 对于大多数结束标签，只是将缓冲的内容放到 document 中。但如果结束标签是</revision>，将会创建一个新的 Put 实例，填入 document 的字段，并提交给表。之后，如果有一段时间没有调用 flushCommits()，就调用它，并向标准输出（put）报告进度。

4.3.4 压缩和 Bloom 过滤器

我们差不多准备好执行这段脚本了，只是还要先做一点清扫工作。列族 text 将包含大块的文本内容，采用某种压缩会有好处。可以启用压缩和快速查找：

```
hbase> alter 'wiki', {NAME=>'text', COMPRESSION=>'GZ', BLOOMFILTER=>'ROW'}
0 row(s) in 0.0510 seconds
```

HBase 支持两种压缩算法：Gzip（GZ）和 Lempel-Ziv-Oberhumer（LZO）。HBase 社区更推荐使用 LZO，这只是一面之辞，但这里要采用 GZ。

LZO 的问题在于实现的许可证。虽然开源，但它不兼容 Apache 的许可证规则，所以 LZO 不能与 HBase 捆绑。在线文档有安装和配置 LZO 支持的详细说明。如果你希望高性能压缩，可以采用 LZO。

Bloom 过滤器是一个很酷的数据结构，它有效地回答了一个问题：我之前见过它吗？Bloom 过滤器最初是由 Burton Howard Bloom 在 1970 年开发的，用于拼写检查，它在数据存储应用中也变得很流行，用于快速确定键是否已存在。如果你不熟悉 Bloom 过滤器，4.3.5 小节有简单的解释，说明了 Bloom 过滤器的工作原理。

HBase 支持使用 Bloom 过滤器，来确定对于某个行键，是否存在某列

（BLOOMFILTER=>'ROWCOL'），或者只是确定某个行键是否存在（BLOOMFILTER=>'ROW'）。列族中列的数量以及行的数量，都是没有限制的。Bloom 过滤器提供了一种快速方法，用于确定数据是否存在，同时避免费时的磁盘读取。

4.3.5　开始

现在我们已经准备好执行这段脚本了。因为前面曾提到这些文件非常大，所以下载和解压缩无疑要花许多时间。那么我们要怎么做？

幸运的是，利用*nix 管道的魔力，可以一步完成下载、解压缩，并将 XML 提供给这段脚本。命令如下：

```
curl <dump_url> | bzcat | \
${HBASE_HOME}/bin/hbase shell import_from_wikipedia.rb
```

Bloom 过滤器是什么原理

我们不深入实现细节，Bloom 过滤器管理一个大小不变的比特数组，初始设置为 0。每次向过滤器提交一段新数据时，某些比特就翻转为 1。对这段数据进行哈希，将哈希值变成一组比特的位置，从而决定哪些比特翻转为 1。

之后，为了测试过滤器是否过去遇到过特定的一段数据，过滤器会算出哪些比特应该是 1，并检查这些比特。如果某个比特是 0，那么过滤器就可以毫不含糊地说没遇到过。如果所有的比特都是 1，那么它报告说遇到过。很可能它以前遇到过这段数据，但随着遇到的数据越来越多，误报的可能性也越来越大。

使用 Bloom 过滤器而不用简单的哈希，是一种折中。哈希不会误报，但存放数据所需的空间是无限的。Bloom 过滤器使用了固定大小的空间，但偶尔会误报，根据饱和程度，误报率是可预测的。

请注意，应该将<dump_url>替换为 WikiMedia 基金会的转储文件的 URL[1]。应该用[project]-latest-pages-articles.xml.bz2 文件，项目可以是英文 Wikipedia（约 6GB）[2]或英文 Wikitionary（约 185MB）[3]。这些文件包含了 Main 名称空间中所有最新的页面版本。也就是说，它们略去了用户页面、讨论页面等。

填入 URL 并执行它！你应该看到这样的输出（最后）：

---

1 http://dumps.wikimedia.org
2 http://dumps.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2
3 http://dumps.wikimedia.org/enwiktionary/latest/enwiktionary-latest-pages-articles.xml.bz2

```
500 records inserted (Ashmore and Cartier Islands)
1000 records inserted (Annealing)
1500 records inserted (Ajanta Caves)
```

只要你愿意，它会很欢乐地执行下去，直到遇到错误。但运行一会儿，你可能希望停止它。如果你打算停止这段脚本，按 Ctrl+C 快捷键。但现在让它保持运行，这样我们就能揭开引擎盖，学习 HBase 如何实现它的横向伸缩性。

4.3.6 区域和监控磁盘使用简介

在 HBase 中，行是按照行键排序的。区域（region）是一组行，由它的起始键（包含）和终止键（排除）来确定。区域不会重叠，每个区域都会指派给集群中一个特定的区域服务器。在简化的独立服务器中，只有一个区域服务器，它将负责所有的区域。完全分布式的集群会包含多个区域服务器。

所以，让我们来看看 HBase 服务器的硬盘使用情况，这会让我们明白数据的分布情况。要检查 HBase 的磁盘使用情况，可以打开一个命令提示符，转到前面指定的 `hbase.rootdir` 目录下，执行 `du` 命令。`du` 是标准的*nix 命令行工具，递归地告诉你一个目录及其子目录使用了多少磁盘空间。选项`-h` 告诉 `du` 用易读的方式来报告数字。

下面是大约插入 12 000 页之后，导入还在进行时，`du` 命令给出的结果：

```
$ du -h
231M    ./.logs/localhost.localdomain,38556,1300092965081
231M    ./.logs
4.0K    ./.META./1028785192/info
12K     ./.META./1028785192/.oldlogs
28K     ./.META./1028785192
32K     ./.META.
12K     ./-ROOT-/70236052/info
12K     ./-ROOT-/70236052/.oldlogs
36K     ./-ROOT-/70236052
40K     ./-ROOT-
72M     ./wiki/517496fecabb7d16af7573fc37257905/text
1.7M    ./wiki/517496fecabb7d16af7573fc37257905/revision
61M     ./wiki/517496fecabb7d16af7573fc37257905/.tmp
12K     ./wiki/517496fecabb7d16af7573fc37257905/.oldlogs
134M    ./wiki/517496fecabb7d16af7573fc37257905
134M    ./wiki
4.0K    ./.oldlogs
365M    .
```

这个输出告诉我们很多关于 HBase 磁盘使用量和分配情况的信息。以`/wiki` 开头的行

描述了 wiki 表的磁盘使用。长名称的子目录 517496fecabb7d16af7573fc37257905 代表了单个区域（目前为止唯一的区域）。在它下面，/text 和/revision 目录分别对应 text 和 revision 列族。最后一行汇总了这些值，告诉我们 HBase 使用了 365M 磁盘空间。

还有一件事值得注意。输出的头两行（以./logs 开头）告诉我们预写式日志（Write-Ahead Log，WAL）文件使用的空间。HBase 利用预写式日志防止节点故障。这是相当典型的灾难恢复技术。例如，文件系统中的预写式日志称为 journaling，在编辑操作（改写和增加）持久到磁盘之前，日志先附加到 WAL。

出于性能考虑，编辑操作不需要马上写入磁盘。如果 I/O 先进行缓冲，随后批量写入磁盘，系统的性能会好很多。在这段状态不定的时间里，如果负责受影响区域的区域服务器崩溃了，HBase 将利用 WAL 来确定哪些操作已成功，并采取纠正操作。

写入 WAL 是可选项，默认是启用的。像 Put 和 Increment 这样的编辑类有一个 setter 方法，名为 setWriteToWAL()，可以用它来排除写入 WAL 的操作。一般情况下，需要保持默认选项，但在某些情况下，改变它是有意义的。例如，如果在执行一项可以随时重新执行导入任务的，就像 Wikipedia 导入脚本一样，那么你可能希望禁用 WAL 写入，牺牲灾难恢复的保护，来换取性能提升。

4.3.7 区域的问讯

如果让这段脚本执行的时间足够长，就会看到 HBase 将该表分成了多个区域。下面是在添加了约 150 000 页之后，du 输出：

```
$ du -h
40K     ./.logs/localhost.localdomain,55922,1300094776865
44K     ./.logs
24K     ./.META./1028785192/info
4.0K    ./.META./1028785192/recovered.edits
4.0K    ./.META./1028785192/.tmp
12K     ./.META./1028785192/.oldlogs
56K     ./.META./1028785192
60K     ./.META.
4.0K    ./.corrupt
12K     ./-ROOT-/70236052/info
4.0K    ./-ROOT-/70236052/recovered.edits
4.0K    ./-ROOT-/70236052/.tmp
12K     ./-ROOT-/70236052/.oldlogs
44K     ./-ROOT-/70236052
48K     ./-ROOT-
138M    ./wiki/0a25ac7e5d0be211b9e890e83e24e458/text
5.8M    ./wiki/0a25ac7e5d0be211b9e890e83e24e458/revision
```

```
4.0K    ./wiki/0a25ac7e5d0be211b9e890e83e24e458/.tmp
144M    ./wiki/0a25ac7e5d0be211b9e890e83e24e458
149M    ./wiki/15be59b7dfd6e71af9b828fed280ce8a/text
6.5M    ./wiki/15be59b7dfd6e71af9b828fed280ce8a/revision
4.0K    ./wiki/15be59b7dfd6e71af9b828fed280ce8a/.tmp
155M    ./wiki/15be59b7dfd6e71af9b828fed280ce8a
145M    ./wiki/0ef3903982fd9478e09d8f17b7a5f987/text
6.3M    ./wiki/0ef3903982fd9478e09d8f17b7a5f987/revision
4.0K    ./wiki/0ef3903982fd9478e09d8f17b7a5f987/.tmp
151M    ./wiki/0ef3903982fd9478e09d8f17b7a5f987
135M    ./wiki/a79c0f6896c005711cf6a4448775a33b/text
6.0M    ./wiki/a79c0f6896c005711cf6a4448775a33b/revision
4.0K    ./wiki/a79c0f6896c005711cf6a4448775a33b/.tmp
141M    ./wiki/a79c0f6896c005711cf6a4448775a33b
591M    ./wiki
4.0K    ./.oldlogs
591M    .
```

最大的改变是老的区域（517496fecabb7d16af7573fc37257905）不见了，代之以 4 个新的区域。在独立服务器中，所有的区域都是由单个服务器提供服务，但在分布式环境下，它们会分包到不同的区域服务器。

这提出了几个问题，如“区域服务器如何知道它们负责为哪些区域提供服务？”以及“你怎样发现哪个区域（以及哪个区域服务器）提供哪一行？”

如果回到 HBase shell，就可以查询.META.表，了解关于目前区域的更多情况。.META.是一个特殊的表，它的唯一目的就是追踪所有的用户表，以及哪个区域服务器负责为这些表的各个区域服务。

```
hbase> scan '.META.', { COLUMNS => [ 'info:server', 'info:regioninfo' ] }
```

即使区域的数目很少，你也会看到大量输出。这里是输出片段，为增加可读性，进行了格式化和截断处理。

```
ROW
 wiki,,1300099733696.a79c0f6896c005711cf6a4448775a33b.

COLUMN+CELL
  column=info:server, timestamp=1300333136393, value=localhost.localdomain:3555
  column=info:regioninfo, timestamp=1300099734090, value=REGION => {
    NAME => 'wiki,,1300099733696.a79c0f6896c005711cf6a4448775a33b.',
    STARTKEY => '',
    ENDKEY => 'Demographics of Macedonia',
    ENCODED => a79c0f6896c005711cf6a4448775a33b,
    TABLE => {{...}}

ROW
 wiki,Demographics of Macedonia,1300099733696.0a25ac7e5d0be211b9e890e83e24e458.
```

```
COLUMN+CELL
   column=info:server, timestamp=1300333136402, value=localhost.localdomain:35552
   column=info:regioninfo, timestamp=1300099734011, value=REGION => {
      NAME => 'wiki,Demographics of Macedonia,1300099733696.0a25...e458.',
      STARTKEY => 'Demographics of Macedonia',
      ENDKEY => 'June 30',
      ENCODED => 0a25ac7e5d0be211b9e890e83e24e458,
      TABLE => {{...}}
```

上面列出的两个区域都由同一个服务器提供服务，localhost.localdomain:35552。第一个区域从空字符串行（''）开始，到'Demographics of Macedonia'结束。第二个区域从'Demographics of Macedonia'开始，到'June 30'。

STARTKEY 是包含在内的，而 ENDKEY 是排除在外的。所以，如果查询'Demographics of Macedonia'这一行，会发现它在第 2 区域中。

因为行是按顺序保存的，所以可以利用保存在.META 中的信息，来查找某一行应该在哪个区域和服务器。但.META.表保存在哪里呢？

我们发现.META.表分割为多个区域，由区域服务器提供服务，就像其他的表一样。要找到哪个服务器提供.META.表的哪一部分，就要扫描-ROOT-。

```
hbase> scan '-ROOT-', { COLUMNS => [ 'info:server', 'info:regioninfo' ] }

ROW
   .META.,,1

COLUMN+CELL
   column=info:server, timestamp=1300333135782, value=localhost.localdomain:35552
   column=info:regioninfo, timestamp=1300092965825, value=REGION => {
      NAME => '.META.,,1',
      STARTKEY => '',
      ENDKEY => '',
      ENCODED => 1028785192,
      TABLE => {{...}}
```

将区域分配给区域服务器，包括分析.META.区域的工作，都是由主节点来处理的，该节点通常称为 HBaseMaster。主服务器也可以是一个区域服务器，同时执行两项任务。

如果一台区域服务器失效，主服务器会介入，重新分配原来属于失效节点的那些区域。这些区域的新服务器会查看 WAL，看看是否需要执行恢复步骤。如果主服务器失效，它的职责将顺延到其他可以成为主服务器的区域服务器。

4.3.8　扫描一个表来建立另一个表

假定你已停止了运行导入的脚本，我们可以转到下一项任务：从导入的 wiki 内容中提取信息。

> **我的表模式在哪里**
>
> 表模式信息从 regioninfo 扫描的示例输出中删除了。这减少了混乱，我们稍后将讨论性能调优选项。如果你很想看到表模式的定义，请使用 describe 命令。下面是例子：
>
> ```
> hbase> describe 'wiki'
> hbase> describe '.META.'
> hbase> describe '-ROOT-'
> ```

wiki 的文法包含许多链接，一些链接到内部的其他文章，一些链接到外部资源。这些内部链接包含了大量的拓扑数据。让我们把它们找出来！

我们的目标是找出文章间的直接链接关系，指向另一篇文章或从另一篇文章得到链接。在 wiki 文本中，内部链接是这样的：*[[<target name>|<alt text>]]*，其中<*target name*>是要链接的文章，<*alt text*>是待显示的替代文本（可选）。

例如，如果关于《星球大战》的文章包含字符串"*[[Yoda|jedi master]]*"，我们希望保存该关系两次：一次是《星球大战》的链出，另一次是 Yoda 的链入。保存该关系两次意味着，寻找一个页面的链入和链出都会很快。

为了保存这种附加的链接数据，要创建一个新表。进入 shell 并输入下面的命令：

```
hbase> create 'links', {
  NAME => 'to', VERSIONS => 1, BLOOMFILTER => 'ROWCOL'
},{
  NAME => 'from', VERSIONS => 1, BLOOMFILTER => 'ROWCOL'
}
```

在原则上，可以选择将链接数据硬挤到原有的列族中，或者在 wiki 表中添加一两个列族，而不是创建一个新表。创建独立的表有一个好处，它会有独立的区域。这意味着在需要的时候，集群可以更有效地分割区域。

根据 HBase 社区对列族给出的一般建议，每个表的列族数目应保持较小。可以将更多的列放到同一个列族中，也可以完全将列族放到不同的表中。这种选择很大程度上取决于客户是否经常需要取得整行数据（而不是只需要几列的值）。

在我们 wiki 的例子中，需要让 text 和 revision 列族放在同一个表里，这样我们在加入新版本时，元数据和文本的时间戳是一样的。但链接数据不一样，它不会与包含它的文章具有相同的时间戳。而且，大多数客户的动作要么对文章内容感兴趣，要到对抽取的文章链接信息感兴趣，不大可能同时对两者感兴趣。所以，将 to 和 from 列族放到独立的表中是有意义的。

4.3.9　构建扫描程序

创建了 links 表之后，就可以编写一段脚本，扫描 wiki 表的所有行。然后，针对每一行，取出 wiki 的文本，解析出链接。最后，针对每个找到的链接，在 links 表中创建链入和链出记录。这段脚本的大分部对你来说应该相当熟悉。大部分内容是重复的，我们将讨论几个新的部分。

```
hbase/generate_wiki_links.rb
import 'org.apache.hadoop.hbase.client.HTable'
import 'org.apache.hadoop.hbase.client.Put'
import 'org.apache.hadoop.hbase.client.Scan'
import 'org.apache.hadoop.hbase.util.Bytes'

def jbytes( *args )
  return args.map { |arg| arg.to_s.to_java_bytes }
end

wiki_table = HTable.new( @hbase.configuration, 'wiki' )
links_table = HTable.new( @hbase.configuration, 'links' )
links_table.setAutoFlush( false )

① scanner = wiki_table.getScanner( Scan.new )

linkpattern = /\[\[([^\[\]\|\:\#][^\[\]\|:]*)(?:\|([^\[\]\|]+))?\]\]/

count = 0

while (result = scanner.next())

② title = Bytes.toString( result.getRow())
text = Bytes.toString( result.getValue( *jbytes( 'text', '' ) ) )

if text

  put_to = nil
③ text.scan(linkpattern) do |target,label|
```

```
      unless put_to
        put_to = Put.new( *jbytes( title ) )
        put_to.setWriteToWAL( false )
      end

      target.strip!
      target.capitalize!

      label = '' unless label
      label.strip!

      put_to.add( *jbytes( "to", target, label ) )

      put_from = Put.new( *jbytes( target ) )
      put_from.add( *jbytes( "from", title, label ) )
      put_from.setWriteToWAL( false )
④     links_table.put( put_from )

      end

⑤     links_table.put( put_to )if put_to

      links_table.flushCommits()

    end

    count += 1
    puts "#{count} pages processed (#{title})" if count % 500 == 0

  end

  links_table.flushCommits()

  exit
```

① 首先，取得一个 Scan 对象，将用它来扫描 wiki 表。

② 提取行和列的数据需要某种字节转换，但通常这也不太困难。

③ 每次页面文本中出现 linkpattern 时，我们提取出目标文章和链接的文本，然后将这些值加入到 Put 实例中。

④ 最后，让表执行累计的 Put 操作。

⑤ 有可能（尽管可能性不大）一篇文章不包含任何链接，所以要用 if put_to 子句。

对这些 Put 操作采用 setWriteToWAL（false）的设置是基于一种判断。因为这个练习是为了学习，而且如果出了什么问题，只要重新运行这个脚本，所以我们希望速度快一些，接受节点可能失效的风险。

4.3.10　运行脚本

如果你准备无视警告，义无反顾，那就执行这段脚本。

```
${HBASE_HOME}/bin/hbase shell generate_wiki_links.rb
```

它应该输出以下内容：

```
500 pages processed (10 petametres)
1000 pages processed (1259)
1500 pages processed (1471 BC)
2000 pages processed (1683)
...
```

像前面的脚本一样，可以让它爱运行多久就运行多久，甚至运行完。如果你想停止，就按 CTRL+C 快捷键。

可以像前面那样，用 du 监控该脚本的磁盘使用情况。你会看到新的项，代表刚创建的 links 表，随着脚本的运行，大小计数会增加。

4.3.11　检查输出

我们刚才以编程方式创建了一个扫描器（scanner），以执行一项复杂的任务。接下来要用 shell 的 scan 命令，简单地将表的部分内容转储到控制台。

针对脚本在 text:blob 中找到的每个链接，它将一视同仁地在 links 表中创建 to 和 from 项。要查看这类创建的链接，可以转到 shell 中，扫描该表。

```
hbase> scan 'links', STARTROW => "Admiral Ackbar", ENDROW => "It's a Trap!"
```

你应该看到许多输出。当然，可以用 get 命令来看某篇文章的链接。

```
hbase> get 'links', 'Star Wars'
COLUMN CELL
...
links:from:Admiral Ackbar                timestamp=1300415922636, value=
links:from:Adventure                     timestamp=1300415927098, value=
```

```
links:from:Alamogordo, New Mexico      timestamp=1300415953549, value=
links:to:"weird al" yankovic           timestamp=1300419602350, value=
links:to:20th century fox              timestamp=1300419602350, value=

links:to:3-d film                      timestamp=1300419602350, value=
links:to:Aayla secura                  timestamp=1300419602350, value=
...
```

Joe 问道：

我们不能用 mapreduce 完成这项任务吗？

4.1 节解释了对应例子将采用（J）Ruby 和 Java Script。JRuby 与 Hadoop 相处得不是太好，但如果你想用 Java 来实现 mapreduce，应该将这段扫描程序写成一个 mapreduce 任务，将它发给 Hadoop。

一般来说，像这样的任务最适合用 mapreduce 来实现。有一些规则格式的输入，由映射程序处理（扫描一个 HBase 表），还有一些输出操作，由规约程序批量处理（将一些行写到一个 HBase 表中）。

Hadoop 架构希望任务实例是用 Java 写的，并完全封装（包括所有依赖关系）到一个 jar 文件中，可以发送到集群的所有节点上。较新版本的 JRuby 可以扩展 Java 类，但与 HBase 一起发布的版本做不到。

有一些开源项目提供了一种桥接办法，能在 Hadoop 上运行 JRuby，但还没有和 HBase 配合得特别好的。有传言说，将来 HBase 的基础结构将包含一些抽象，让 JRuby 的映射规约任务成为可能。所以，将来还是有希望的。

在 wiki 表中，行比列更有规则。前面提到，每行都有 text:、revision:author 和 revision:comment 等列。links 表没有这样规则。每行可以有一列或几百列。列名的多样性与行键本身（维基百科文章的标题）的多样性一样。这不是什么问题！HBase 称为稀疏数据存储，正是出于这个原因。

要查出表中现在有多少行，可以用 count 命令。

```
hbase> count 'wiki', INTERVAL => 100000, CACHE => 10000
Current count: 100000, row: Alexander wilson (vauxhall)
Current count: 200000, row: Bachelor of liberal studies
Current count: 300000, row: Brian donlevy
...
Current count: 2000000, row: Thomas Hobbes
Current count: 2100000, row: Vardousia
Current count: 2200000,row: W��rrstadt (verbandsgemeinde)
```

```
2256081 row(s) in 173.8120 seconds
```

由于 HBase 的分布式架构，它不能立即知道每个表中有多少行。要查出结果，必须对它们进行计数（执行表扫描）。幸运的是，HBase 的存储架构是基于区域的，这让它能够进行快速的分布式扫描。所以，即使操作需要表扫描，我们也不需要像其他数据库那样担心。

4.3.12 第 2 天总结

真是充实的一天！我们学习了如何为 HBase 编写导入脚本，这段脚本从 XML 流中解析出数据。然后运用这些技巧，直接将维基百科的数据转储到 `wiki` 表中。

我们学习了更多的 HBase API，包括一些客户端可以控制的性能级别，如 `setAutoFlush()`、`flushCommits()`和 `setWriteToWAL()`。伴随这些代码行，我们讨论了 HBase 的一些架构特性，如灾难恢复，这是通过预写式日志来提供的。

说到架构，我们发现了表的区域，以及 HBase 如何将职责分摊给集群中的各个区域服务器。我们扫描了`.META.`和`-ROOT-`表，目的是感受 HBase 的内部结构。

最后，我们讨论了 HBase 的稀疏设计在性能方面的隐含意义。同时，我们介绍了关于列、列族和表的一些社区最佳使用实践。

第 2 天作业

求索

- 找到有关 HBase 中压缩的优缺点的讨论或文章。
- 找到一篇文章，解释 Bloom 过滤器的一般工作原理，以及对 HBase 产生怎样的好处。
- 除了使用哪个算法之外，列族还有其他哪些选项与压缩有关？
- 数据类型和预期的使用方式是如何先知列族的压缩选项的？

实践

在数据导入的想法上进行扩展，让我们来创建一个包含营养成分信息的数据库。

从 Data.gov 下载 MyPyramid Raw Food Data 集[1]。解压缩，找到 `Food_ Display_`

[1] http://explore.data.gov/Health-and-Nutrition/MyPyramid-Food-Raw-Data/b978-7txq

Table.xml。

这些数据包含许多对<Food_Display_Row>标签。在这些标签里，每行都有一个<Food_Code>（整数值），一个<Display_Name>（字符串），还有一些其他的实际食品数据，包含在相应名称的标签中。

- 创建一个名为 foods 的新表，它只包含一个列族，存放这些实际数据。你用什么作为行键？对这些数据，采用怎样的列族选项比较有意义？
- 创建一段新的 JRuby 脚本来导入食品数据。采用前面 Wikipedia 导入脚本中用到的 SAX 解析方式，调整后用于食品数据。
- 在命令行里，将食品数据传递给导入脚本，填充 foods 表。
- 最后，通过 HBase 的 shell，查询你最喜爱的食品的信息。

4.4 第 3 天：放入云端

在前 2 天，我们在单机模式下使用了 HBase，获得了许多一手的经验。到目前为止，我们的经验集中于访问单个本地数据库。实际上，如果选择使用 HBase，你想要的是一个有相当规模的集群，从而实现分布式架构的性能提升。

在第 3 天，我们将关注点转向远程 HBase 集群的操作与交互。首先，要用 Ruby 开发一个客户端应用，它采用二进制协议 Thrift 来连接本地服务器。然后利用 Amazon EC2 提供的云服务，启动一个多节点的集群，并采用 Apache Whirr 作为集群管理技术。

4.4.1 开发 Thrift 协议的 HBase 应用

到目前为止，我们一直在使用 HBase shell，但 HBase 支持多种客户端连接协议。下面是完整的列表：

名　称	连 接 方 法	是否可用于产品
Shell	直接	是
Java API	直接	是
Thrift	二进制协议	是
REST	HTTP	是
Avro	二进制协议	否（仍在试验阶段）

在上面的列表中，连接方法描述了该协议是直接调用 Java，还是通过 HTTP 传送数据，

或是用紧凑的二进制格式传送数据。除 Avro 之外，它们都是产品级的。Avro 还比较新，应视为试验特性。

在所有这些选择中，Thrift 可能是开发客户端应用时最常采用的。Thrift 是一个成熟的二进制协议，开销很小，最初由 Facebook 开发并开放源代码，后来成为 Apache 孵化项目。让我们在机器上做好连接 Thrift 的准备。

1. 安装 Thrift

像数据库领域上的许多事情一样，使用 Thrift 需要一些设置。要通过 Thrift 连接 HBase 服务器，需要完成下列工作。

1. 让 HBase 运行 Thrift 服务。
2. 安装 Thrift 命令行工具。
3. 安装选定客户端语言的库。
4. 针对你的语言生成 HBase model 文件。
5. 创建并运行客户端应用。

从运行 Thrift 服务开始，这相当容易。像这样从命令行启动守护进程：

```
${HBASE_HOME}/bin/hbase-daemon.sh start thrift -b 127.0.0.1
```

接下来，需要安装 Thrift 命令行工具。所需的步骤主要取决于具体环境，一般需要编译。要检查安装是否正确，可以在命令行里带-version 参数运行。你应该看到类似这样的输出：

```
$ thrift -version
Thrift version 0.6.0
```

采用 Ruby 作为客户端应用的语言，但其他语言的操作步骤也是类似的。在命令行安装 Thrift Ruby gem 是这样的：

```
$ gem install thrift
```

要检查 gem 是否正确安装，可以运行这样一行 Ruby 代码：

```
$ ruby -e "require 'thrift'"
```

如果你没看到输出信息，即就成功了！如果看到以“no such file to load”开

始的出错消息，就意味着你应该停下来，解决问题后再继续。

2. 生成 model 文件

接下来，要生成针对语言的一些 HBase model 文件。这些 model 文件为特定的 HBase 版本和你安装的具体 Thrift 版本建立联系，所以必须生成（而不是预先准备好）它们。首先，定位${HBASE_HOME}/src 目录下的 Hbase.thrift 文件。路径应该类似这样：

```
${HBASE_HOME}/src/main/resources/org/apache/hadoop/hbase/thrift/Hbase.thrift
```

确定了路径后，用下面的命令来生成 model 文件，尖括号部分用路径取代：

```
$ thrift --gen rb <path_to_Hbase.thrift>
```

这将创建新的文件夹，名为 gen-rb，其中包含下列 model 文件：

- hbase_constants.rb
- hbase.rb
- hbase_types.rb

接下来，在构建简单的客户端应用时，会用到这些文件。

3. 构建客户端应用

应用程序将通过 Thrift 协议连接 HBase，然后列出它找到的所有表，以及表中的列族。这些是为 HBase 开发管理界面的前几步。不像前面的那些例子，这段脚本将在普通的 Ruby 下运行，而不是 JRuby。例如，它可以包含在 Ruby 开发的 Web 应用中。

将这段脚本录入到一个新文本文件中（称为 thrift_example.rb）：

```
hbase/thrift_example.rb
$:.push('./gen-rb')
require 'thrift'
require 'hbase'

socket = Thrift::Socket.new( 'localhost', 9090 )
transport = Thrift::BufferedTransport.new( socket )
protocol = Thrift::BinaryProtocol.new( transport )
client = Apache::Hadoop::Hbase::Thrift::Hbase::Client.new( protocol )

transport.open()

client.getTableNames().sort.each do |table|
```

```
    puts "#{table}"
    client.getColumnDescriptors( table ).each do |col, desc|
      puts " #{desc.name}"
      puts " maxVersions: #{desc.maxVersions}"
      puts " compression: #{desc.compression}"
      puts " bloomFilterType: #{desc.bloomFilterType}"
    end
  end

  transport.close()
```

在上面的代码中，第一件事是将 `gen-rb` 加入到路径中，确保 Ruby 能找到那些 model 文件，并包含 `thrift` 和 `hbase`。

在打开 `transport` 后，利用 `getTableNames()` 取回所有表名并进行迭代，针对每个表，利用 `getColumnDescriptors()` 取回所有列族进行迭代，并将一些属性输出到标准输出。

现在，在命令行运行这个程序。输出应该看起来差不多，因为正在连接前面启动的本地 HBase 数据库。

```
$> ruby thrift_example.rb
links
  from:
    maxVersions: 1
    compression: NONE
    bloomFilterType: ROWCOL
  to:
    maxVersions: 1
    compression: NONE
    bloomFilterType: ROWCOL
  wiki
    revision:
      maxVersions: 2147483647
      compression: NONE
      bloomFilterType: NONE
    text:
      maxVersions: 2147483647
      compression: GZ
      bloomFilterType: ROW
```

你会发现，针对 HBase 的 Thrift API 几乎与前面使用的 Java API 具有同样的功能，但

许多概念的表达方式不同。例如，在 Thrift 中没有创建 Put 实例，而是创建一个 Mutation 来更新单个列，或创建一个 BatchMutation，在一个事务中更新多个列。

前面利用 HBase.thrift 文件生成了一些 model 文件（参见 4.4.1 小节下面的“2. 生成 model 文件”节），它本身包含了许多很好的文档，说明了可用的结构和方法。请参考一下！

4.4.2 Whirr 简介

利用云服务建立可用的集群，过去需要很多工作。幸运的是，Whirr 正在改变这种情况。Whirr 目前还是 Apache 的孵化项目，它提供了一些工具，实现虚拟机集群的启动、连接和销毁。它支持一些流行的服务，如 Amazon 的 Elastic Compute Cloud（EC2）和 RackSpace 的云服务器。Whirr 目前支持设置 Hadoop、HBase、Cassandra、Voldemort 和 ZooKeeper 集群，对 MongoDB 和 ElasticSearch 等更多技术的支持正在进行中。

虽然像 Amazon 这样的服务提供商通常提供一些方法，在虚拟机终止后保存一些数据，但我们不会使用这些方法。对我们来说，临时的集群，在终止后会丢失所有数据，就已经足够了。如果你决定将来在生产环境中使用 HBase，你可能希望建立持久存储。如果是这样，值得考虑一下专门的硬件，也许更适合你的需求。像 EC2 这样的动态服务对于即时变化的计算能力非常合适，但专门的物理或虚拟机集群通常更划算。

4.4.3 设置 EC2

在使用 Whirr 来管理集群之前，需要一个云服务提供商的账号和支持。本章介绍如何使用 Amazon 的 EC2，但可以选用其他提供商。

如果你还没有 Amazon 账号，请到 Amazon 的 Web 服务（AWS）门户上获取一个。[1]登录，如果你的账号还没有激活，就对你的账号启用 EC2。[2]最后，在 Accounts Amazon EC2 下打开 EC2 AWS 的控制台页面。[3]它看起来应该如图 4-5 所示。

你需要 AWS 证书（credential）来启动 EC2 的节点。回到 AWS 的主页，选择 Account Security Credentials，移到下面的 Access Credentials 部分，记下你的 Access Key ID。在 Secret Access Key 下面，点击 Show，也请记下这个值。在稍后配置 Whirr 时，分别称这些键为 AWS_ACCESS_KEY_ID 和 AWS_SECRET_ACCESS_KEY。

---

1 http://aws.amazon.com/
2 http://aws.amazon.com/ec2/
3 https://console.aws.amazon.com/ec2/#s=Instances

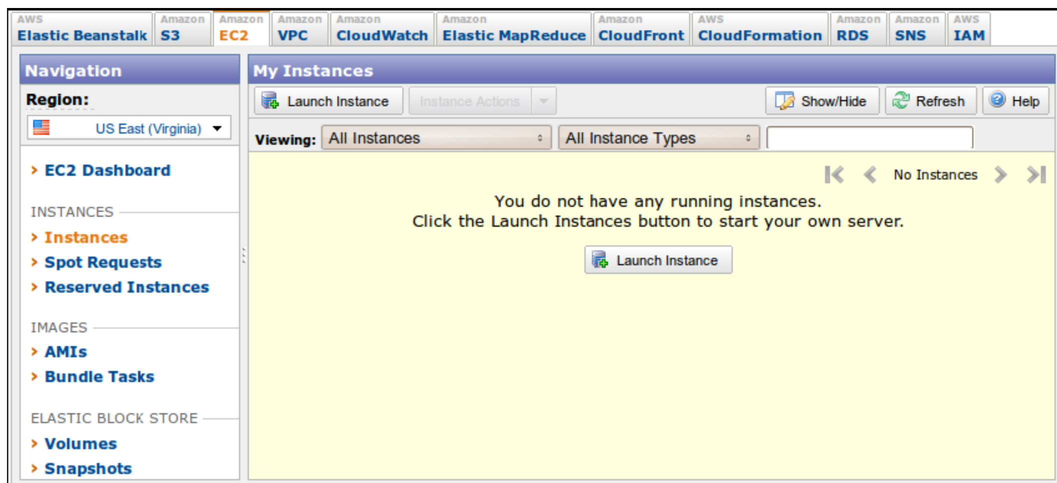


图 4-5　Amazon EC2 控制台，无实例

4.4.4　准备 Whirr

有了 EC2 证书，让我们来获取 Whirr。到 Apache Whirr 网站[1]下载最新的版本。解压下载的文件，然后打开一个命令行提示符，转到这个目录下。可以执行 version 命令，检查 Whirr 是否已经准备好了。

```
$ bin/whirr version
Apache Whirr 0.6.0-incubating
```

接下来，将为 Whirr 创建一些无口令的 SSH 键，用于启动实例（虚拟机）。

```
$ mkdir keys
$ ssh-keygen -t rsa -P '' -f keys/id_rsa
```

这会创建一个名为 keys 的目录，并在里面生成一个 id_rsa 文件和一个 id_rsa.pub 文件。解决了这些细节问题后，就可以开始配置集群了。

4.4.5　配置集群

为了指定关于集群的细节，向 Whirr 提供一个.properties 文件，其中包含相关的设定。在 Whirr 目录下创建一个名为 hbase.properties 的文件，内容如下（在标明的

[1] http://incubator.apache.org/whirr/

地方插入 AWS_ACCESS_KEY_ID 和 AWS_SECRET_ACCESS_KEY）：

```
hbase/hbase.properties
# service provider
whirr.provider=aws-ec2
whirr.identity=your AWS_ACCESS_KEY_ID here
whirr.credential=your AWS_SECRET_ACCESS_KEY here

# ssh credentials
whirr.private-key-file=keys/id_rsa
whirr.public-key-file=keys/id_rsa.pub

# cluster configuration
whirr.cluster-name=myhbasecluster
whirr.instance-templates=\
   1 zookeeper+hadoop-namenode+hadoop-jobtracker+hbase-master,\
   5 hadoop-datanode+hadoop-tasktracker+hbase-regionserver

# HBase and Hadoop version configuration
whirr.hbase.tarball.url=\
   http://apache.cu.be/hbase/hbase-0.90.3/hbase-0.90.3.tar.gz
whirr.hadoop.tarball.url=\
   http://archive.cloudera.com/cdh/3/hadoop-0.20.2-cdh3u1.tar.gz
```

程序的前两部分指明了服务提供商和所有相关的证书（主要是样板代码），后两部分是具体针对要创建的 HBase 集群的。其中 whirr.cluster-name 并不重要，除非你打算同时运行多个集群。不同的集群需要不同的名称。属性 whirr.instance-templates 包含一个逗号分隔的列表，描述各节点承担什么角色，以及有多少个节点。在例子中，希望有一个主服务器和 5 个区域服务器。最后，whirr.hbase.tarball.url 强制 Whirr 使用同样版本的 HBase，也是我们一直在使用的版本。

4.4.6 启动集群

所有配置细节保存到 hbase.properties 中之后，就可以启动集群了。在命令行中，在 Whirr 目录下，执行 launch-cluster 命令，以刚才生成的属性文件作为参数。

```
$ bin/whirr launch-cluster --config hbase.properties
```

这会产生许多输出，并且需要等待一些时间。你可以回到 AWS EC2 的控制台，查看启动的进度。它看起来应该如图 4-6 所示。

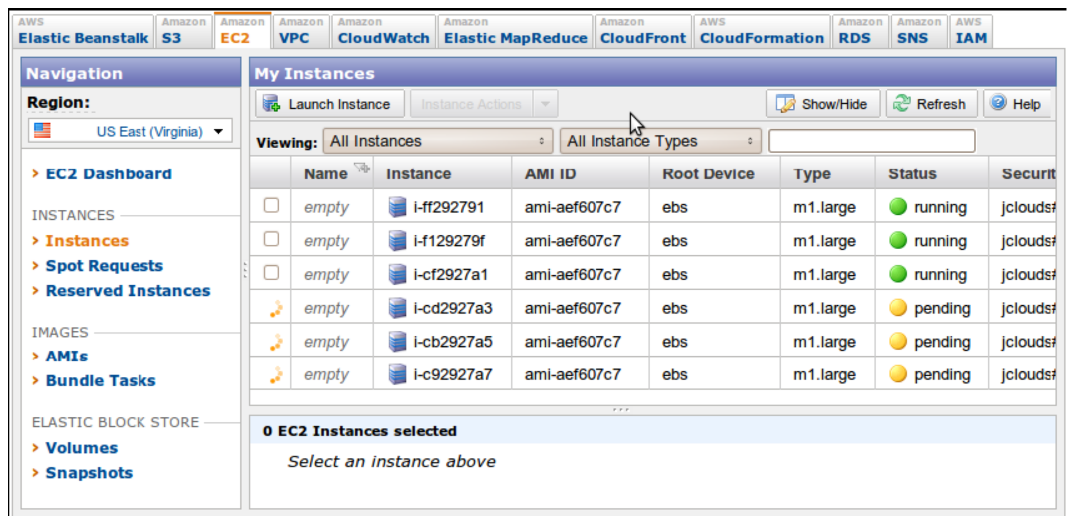


图 4-6　Amazon EC2 控制台显示 HBase 实例正在启动

关于启动状态的更多信息，可以查看 Whirr 目录下的 `whirr.log`。

4.4.7　连接集群

默认情况下，只允许与集群的安全通信，所以要连接 HBase，需要打开 SSH 会话。首先，需要知道要连接的集群中服务器的名称。在你的用户根目录下，Whirr 创建了一个名为 `.whirr/myhbasecluster` 的目录。在这个目录中，你会看到一个以制表符（tab）分隔的文件，名为 `instances`，其中列出了该集群所有正在运行的 Amazon 实例。第 3 列是服务器公开可访问的域名。把第一个域名代入下面的命令：

```
$ ssh -i keys/id_rsa ${USER}@<SERVER_NAME>
```

连接成功后，启动 HBase shell：

```
$ /usr/local/hbase-0.90.3/bin/hbase shell
```

在 shell 启动之后，可以通过 `status` 命令来检查集群的状态。

```
hbase> status
6 servers, 0 dead, 2.0000 average load
```

此后，就可以执行第 1 天和第 2 天中执行过的所有操作，如创建表和插入数据等。用示例的 Thrift 客户端应用连接集群，留在作业中练习。

当然，结束今天的学习之前还有一件事值得讨论，即销毁集群。

4.4.8 销毁集群

完成了远程 HBase EC2 集群之后，利用 Whirr 的 destroy-cluster 命令来关闭它。请注意，这样做会丢失所有插入集群中的数据，因为没有配置这些实例以使用持久存储。

在命令提示符下，在 Whirr 目录中，执行下面的命令：

```
$ bin/whirr destroy-cluster --config hbase.properties
Destroying myhbasecluster cluster
Cluster myhbasecluster destroyed
```

这应该只需要一会儿工夫。在 AWS 控制台中确认这些实例已关闭，如图 4-7 所示。

如果在关闭这些实例时出错，要记得也可以通过 AWS 控制台直接关闭它们。

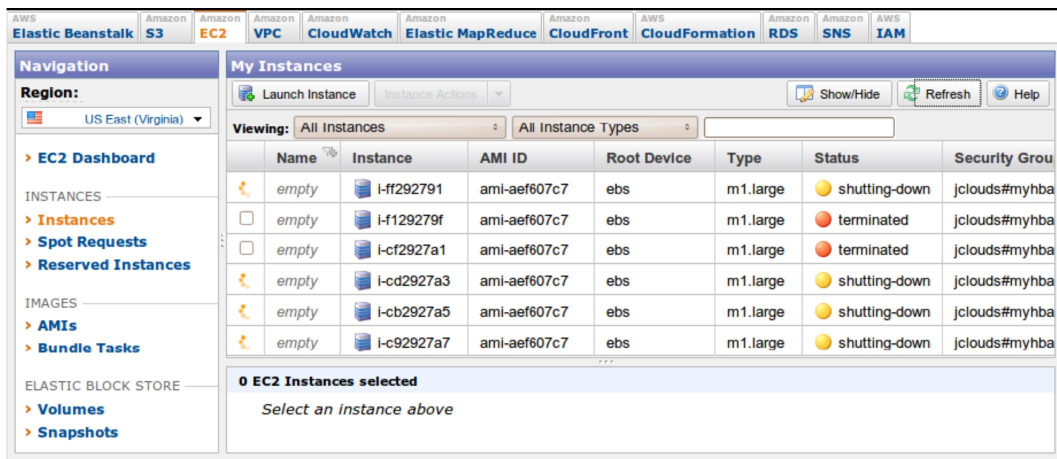


图 4-7　Amazon EC2 控制台显示 HBase 实例正在关闭

4.4.9 第 3 天总结

今天，我们离开了 HBase shell，考察了其他的连接方式，包括二进制协议 Thrift。我们开发一个 Thrift 客户端应用，然后通过 Apache Whirr，在 Amazon EC2 中创建并管理一个远程集群。在今天的作业中，你要将这两步串起来，通过在本地运行的 Thriff 应用，查询远程的 EC2 集群。

第 3 天作业

在今天的作业中，你要用本地的 Thrift 应用来连接远程运行的 HBase 集群。要做到这一点，先要打开集群，允许不安全的 TCP 接入。如果是在生产环境中，先为 Thrift 创建一个安全通道会更好，例如，在 EC2 和本地网络之间建立虚拟专用网（Virtual Private Network，VPN）。这样的设置超出了本书的范围，我们需要说的是，在必要的时候，强烈建议确保通信安全。

实践

- 在你的 EC2 集群运行时，打开 SSH 会话连接一个节点，启动 Hbase shell，然后创建一个表，其中至少包含一个列族。
- 在同一个 SSH 会话中，启动 Thrift 服务。

```
$ sudo /usr/local/hbase-0.90.3/bin/hbase-daemon.sh start thrift -b 0.0.0.0
```

- 利用 Amazon EC2 的 Web 界面控制台，在 Security Groups 中，为集群打开 TCP 端口 9090（Network & Security→Security Groups→Inbound→Create a new rule）。
- 修改前面开发的、基于 Thrift 的简单 Ruby 客户端应用，连接到运行 Thift 的 EC2 节点，而不是本地主机。运行程序，确认它显示了刚创建的表的正确信息。

4.5 总结

HBase 是简单性和复杂性的统一体。数据存储模型相当简单，有一些内置的模式限制。许多术语来自关系数据库，但改变了原来的意义，这对学习帮助不大（例如，术语表和列）。大部分 HBase 模式设计决定表和列的性能特点。

4.5.1 HBase 的优点

HBase 中值得注意的特性包括健壮的可伸缩架构，以及内置的版本和压缩功能。HBase 内置的版本功能在某些情况下是强大的特性。例如，对于监控和维护来说，保存 wiki 页面的版本历史是一项关键特性。选择了 HBase，就不必通过其他方法来实现页面历史，免费实现了这一特征。

在性能方面，HBase 意味着可伸缩。如果你有大量的数据，达到很多 GB 或很多 TB，HBase 可能适合你。HBase 是支持机架的，在数据中心的机架内或机架之间复制数据，这样就能够优雅而快速地处理节点失效。

HBase 社区相当不错。几乎总是有人在 IRC 频道[1]或邮件列表[2]上准备回答问题，给你指出正确的方向。虽然有一些知名度很高的公司在项目中使用 HBase，但没有公司提供 HBase 服务。这意味着，HBase 社区的人这么做是出于对项目的热爱和公共利益。

4.5.2 HBase 的缺点

虽然 HBase 的设计目的是可伸缩性，但它不能缩小。HBase 社区似乎一致同意，5 个节点是你想要的最小配置。因为它的设计目的是大集群，所以它也较难管理。解决小问题不是 HBase 关心的事情，非专家级的文档难以得到，这使得学习曲线更陡。

另外，HBase 几乎从来不会单独部署。它是一些可伸缩模块构成的生态系统中的一部分。这包括 Hadoop（Google 的 MapReduce 思想的一种实现）、Hadoop 分布式文件系统（HDFS）和 Zookeeper（一种无领导者的服务（headless service），帮助节点间的协调）。这个生态系统既有优势，又有劣势。它提供强大的架构稳定性，同时也给管理员带来了更繁重的维护工作。

HBase 有一个值得一提的特点，即除了行键之外，它不提供任何索引功能。行是按行键排序来保存的，但其他字段没有这样的排序，如列名和值。所以，如果你希望不按行键来查找行，就需要扫描表，或自己维护索引。

另一个缺失的概念是数据类型。HBase 中所有字段的值都作为不解释的字节数组。比如，整数值、字符串和日期之间没有区别。对 HBase 来说，它们都是一些字节，所以这些字节的解释取决于应用程序。

4.5.3 HBase on CAP

在 CAP（Consistency、Availability、Partition Tolerance，即一致性、可用性和分区容错性）方面，HBase 肯定是 CP。HBase 提供强大的一致性保证。如果某个客户端成功地写入了一个值，其他客户端将在接下来的请求中收到这个更新的值。有些数据库（如 Riak）允许你以每个操作为基础来实现 CAP。HBase 不是这样的。在面对合理数量的分区时（如一个节点失效），HBase 仍然可用，它会将职责转到集群中的其他节点。但在严重故障情况下，只有一个节点还有效，HBase 就别无选择，只能拒绝请求。

如果引入集群到集群的复制，关于 CAP 的讨论就会有点复杂。这是本章没有介绍的高级功能。典型的多集群设置会有几个集群，分布在有一定距离的不同位置。在这种情况下，

---

[1] irc://irc.freenode.net/#hbase

[2] http://hbase.apache.org/mail-lists.html

对于给定的列族，一个集群负责记录，其他集群只是提供对复制数据的访问。系统最终会一致，因为复制集群将提供它们认为的最新数据，但可能不是主集群中最新的数据。

4.5.4　结束语

作为我们遇到的第一个非关系数据库，HBase 相当有挑战。术语具有欺骗性，让我们觉得好像轻松，而实际上安装和配置 HBase 需要内心比较强大才行。在好的一面，HBase 提供了一些非常独特的特性，如版本和压缩。在解决一些特定问题时，这些方面让 HBase 很有吸引力。当然，它的伸缩性相当好，可以在常用硬件上扩展到许多节点。总的来说，HBase 就像射钉枪，是相当强大的工具，所以要当心你的手指。

第 5 章

MongoDB

MongoDB 在许多方面就像一个电钻。完成任务的能力主要取决于选用的组件（从不同尺寸的钻头到磨砂机适配器（sander adapter））。MongoDB 的长处在于多功能、强大、易于使用，而且既能完成大任务，也能完成小任务。虽然它是一个很新的工具，但逐渐成为工程师经常采用的工具。

MongoDB 于 2009 年首次发布，成为 NoSQL 领域中冉冉升起的一颗新星。它被设计成为一个可伸缩的数据库（Mongo 的名字来自于“humongous”，其大无比），性能和易于存取数据是其核心设计目标。它是一个文档数据库，允许数据以嵌套的状态保存，而且重要的是，它能够以任意方式查询嵌套的数据。它不强制使用模式（与 Riak 相似，不同于 Postgres），所以文档选择性包含的一些字段或类型，在该集合的其他文档中可以没有。

但不要认为 MongoDB 的灵活性让它变成一个玩具。有一些规模巨大的 MongoDB（常简称为 Mongo）生产环境部署，如 Foursquare、bit.ly，CERN 也用它来收集大型强子对撞机的数据。

5.1 其大无比

关系数据库有强大的查询能力，Riak 和 HBase 这样的数据存储具有分布式的特点，Mongo 在这两者之间找到了最佳结合点。项目创始人 Dwight Merriman 曾说过，MongoDB 就是他希望在 DoubleClick 时拥有的数据库，那时他是 DoubleClick 的 CTO，需要保存大规模的数据，又要能满足自由定义的查询。

Mongo 是一个 JSON 文档数据库（虽然在技术上数据是以二进制 JSON 的形式保存的，即所谓的 BSON）。Mongo 文档可以看成是没有模式的关系表，它的值可以嵌套任意深度，

要理解 JSON 文档的概念，请看图 5-1。

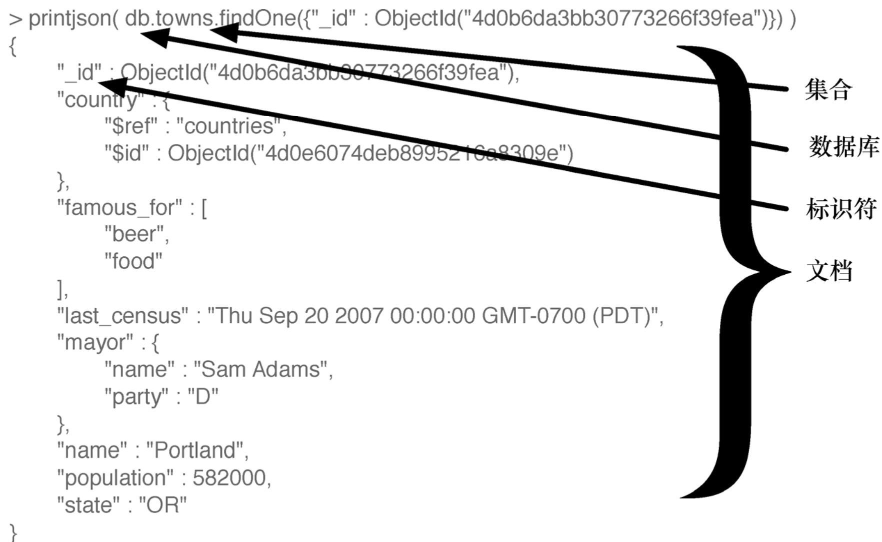


图 5-1　Mongo 文档转出为 JSON 的形式

一些 Web 项目需要高伸缩性的数据存储，但预算很少，不能购卖“大铁块”似的昂贵硬件。这样的项目越来越多，Mongo 正是极好的选择。正因为 Mongo 缺乏结构化的模式，它可以随着数据模型而增长和变化。如果你就职于一个做 Web 项目的创业公司，怀着做大的梦想，或者已经很大，需要扩展服务器的数量，请考虑使用 MongoDB。

5.2　第 1 天：CRUD 和嵌套

我们今天将学习一些CRUD操作，最后完成在MongoDB中进行嵌套查询。像往常一样，我们不会展示安装步骤，但如果你访问Mongo网站[1]，就可以下载适合你的操作系统的版本，或找到从源代码构建的说明。如果你使用OS X，我们推荐通过Homebrew安装（`brew install mongodb`）。如果你使用Debian/Ubuntu的变种，请试用Mongodb.org自己的apt-get包。

为了防止录入错误，Mongo 要求你先创建目录，让 `mongod` 来存放数据。常见的位置是`/data/db`。要确保运行服务器程序的用户对这个目录有读写权限。如果它还没有运行，可以通过运行 `mongod` 来启动 Mongo 服务。

---

[1] http://www.mongodb.org/downloads

5.2.1 命令行的乐趣

要创建一个名为 book 的新数据库，先在终端上执行下面的命令。它将连接到一个命令行界面，该界面是模仿 MySQL 的。

```
$ mongo book
```

在控制台输入 help，这是不错的开始。我们目前在 book 数据库中，但你可以通过 show dbs 来查看其他数据库，通过 use 命令切换数据库。

在 Mongo 中创建一个集合（collection，类似于 Riak 术语中的 bucket）非常容易，只要在该集合中加入第一条记录。因为 Mongo 是没有模式的，所以不需要先定义任何东西，使用它就够了。而且，如果不在 book 数据库中添加值，它实际上并不存在。下面的代码创建/插入了一个 towns 集合：

```
> db.towns.insert({
  name: "New York",
  population: 22200000,
  last_census: ISODate("2009-07-31"),
  famous_for: [ "statue of liberty", "food" ],
  mayor : {
    name : "Michael Bloomberg",
    party : "I"
  }
})
```


Eric 说：

观望

在我改变自己的产品代码之前，我对使用文档式的数据存储库持观望态度。我来自于关系数据库的世界，我发现迁移到 Mongo 很容易，因为它可以即席（ad hoc）查询。而且它的伸缩能力也符合我的 Web 伸缩梦想。但除了结构，我更信任开发团队。他们很乐意承认 Mongo 不完美，但他们的计划很清晰（而且通常遵守这些计划），这些计划是基于一般 Web 架构使用场景，而不是理想化地争论伸缩性和复制。在用 MongoDB 时，这种对可用性的务实关注会闪闪发光。对这种渐进行为的折中，就是在 Mongo 中任何给定的功能，都有几种方式来执行。

前一节说过文档是 JSON（实际上是 BSON），所以以 JSON 的格式添加新文档，其中的花括号{...}表示一个对象（如一个哈希表或映射表），对象包含键和值。其中的方括号[...]表示一个数组。可以用任意深度嵌套这些值。

通过 show collections 命令，可以检验现在存在的集合。

```
> show collections

system.indexes
towns
```

刚刚创建了 towns，而 system.indexes 已经存在了。可以通过 find() 列出一个集合的内容。为了可读，对输出进行了格式化，但输出可能就是带换行的一行。

```
> db.towns.find()

{
  "_id" : ObjectId("4d0ad975bb30773266f39fe3"),
  "name" : "New York",
  "population": 22200000,
  "last_census": "Fri Jul 31 2009 00:00:00 GMT-0700 (PDT)",
  "famous_for" : [ "statue of liberty", "food" ],
  "mayor" : { "name" : "Michael Bloomberg", "party" : "I" }
}
```

与关系数据库不同，Mongo 不支持服务器端的连接（Join）操作。单条 JavaScript 调用会取出一个文档，以及它所有嵌套的内容，不需要额外的工作。

你可能已经注意到，新插入城镇的 JSON 输出包含一个 _id 字段，值是 ObjectId。这和 PostgreSQL 中 SERIAL 产生自增的数字主键类似。ObjectId 总是 12 字节，由时间戳、客户端机器 ID、客户端进程 ID 和 3 字节的增量计数器组成。图 5-2 展示了这些字节的布局。

这种自动计数设计的好处在于，每台机器上的每个进程都能够处理自己的 ID 生成，而不会与其他 mongod 实例发生冲突。这个设计选择提示了 Mongo 的分布式特征。

JavaScript

Mongo 的母语是 JavaScript，不论是复杂到 mapreduce 查询，还是简单到要求帮助。

```
> db.help()
> db.towns.help()
```

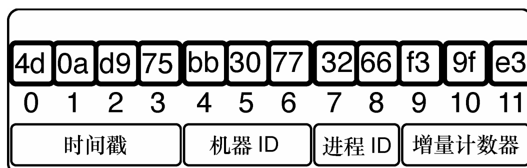


图 5-2　ObjectId 布局示例

这些命令将针对给定对象，列出相关的可用函数。db 是一个 JavaScript 对象，它包含当前数据库的有关信息。db.x 是一个 JavaScript 对象，代表一个集合（名为 x）。命令就是 JavaScript 函数。

```
> typeof db
object
> typeof db.towns
object
> typeof db.towns.insert
function
```

如果你想查看源代码，就不带参数和圆括号来调用它（更像 Python，而非 Ruby）。

```
db.towns.insert
function (obj, _allow_dot) {
    if (!obj) {
        throw "no object passed to insert!";
    }
    if (!_allow_dot) {
        this._validateForStorage(obj);
    }
    if (typeof obj._id == "undefined") {
        var tmp = obj;
        obj = {_id:new ObjectId};
        for (var key in tmp) {
            obj[key] = tmp[key];
        }
    }
    this._mongo.insert(this._fullName, obj);
    this._lastID = obj._id;
}
```

让我们创建自己的 JavaScript 函数，向 towns 集合填充更多的文档。

```
mongo/insert_city.js
function insertCity(
  name, population, last_census,
  famous_for, mayor_info
) {
  db.towns.insert({
    name:name,
    population:population,
    last_census: ISODate(last_census),
    famous_for:famous_for,
    mayor : mayor_info
```

```
    });
}
```

可以将代码复制到 shell。然后可以调用它。

```
insertCity("Punxsutawney", 6200, '2008-31-01',
    ["phil the groundhog"], { name : "Jim Wehrle" }
)

insertCity("Portland", 582000, '2007-20-09',
    ["beer", "food"], { name : "Sam Adams", party : "D" }
)
```

集合中现在应该有了 3 个城镇，可以像以前那样调用 db.towns.find()来确认。

5.2.2　Mongo的更多有趣内容

前面我们曾不带参数调用 find()函数，取得所有文档。要访问特定的文档，只需要设置_id 属性。_id 是 ObjectId 类型，要查询，就必须利用 ObjectId（str）函数，对字符串进行转换。

```
db.towns.find({ "_id" : ObjectId("4d0ada1fbb30773266f39fe4") })

{
  "_id" : ObjectId("4d0ada1fbb30773266f39fe4"),
  "name" : "Punxsutawney",
  "population" : 6200,
  "last_census" : "Thu Jan 31 2008 00:00:00 GMT-0800 (PST)",
  "famous_for" : [ "phil the groundhog" ],
  "mayor" : { "name" : "Jim Wehrle" }
}
```

find()函数也接受可选的第二个参数，它是一个字段对象，用于过滤要取得哪些字段。如果只需要城镇的名称（以及_id），传入 name 和结果为 1（或 true）的值。

```
db.towns.find({ _id : ObjectId("4d0ada1fbb30773266f39fe4") }, { name : 1 })
{
  "_id" : ObjectId("4d0ada1fbb30773266f39fe4"),
  "name" : "Punxsutawney"
}
```

要取得除名称外的所有字段，将 name 设置为 0（或者 false 或 null）。

```
db.towns.find({ _id : ObjectId("4d0ada1fbb30773266f39fe4") }, { name : 0 })
```

```
{
  "_id" : ObjectId("4d0ada1fbb30773266f39fe4"),
  "population" : 6200,
  "last_census" : "Thu Jan 31 2008 00:00:00 GMT-0800 (PST)",
  "famous_for" : [ "phil the groundhog" ]
}
```

像PostgreSQL一样，在Mongo中可以构造自由定义的查询，按字段的值、范围或组合条件来查询。要找到以字母P开头、人口少于 10 000 的所有城镇，可以使用Perl兼容的正则表达式（PCRE）[1]和一个范围操作符。

```
db.towns.find(
  { name : /^P/, population : { $lt : 10000 } },
  { name : 1, population : 1 }
)

{ "name" : "Punxsutawney", "population" : 6200 }
```

Mongo 中的条件操作符遵循字段的格式：`{ $op : value }`，其中`$op`是一个操作符，如`$ne`（不等于）。你可能想要更简明的语法，如 `field<value`。但这是 `JavaScript` 代码，不是领域特定的查询语言，所以查询必须符合 JavaScript 的语法规则（今天晚些时候我们会看到，在特定情况下如何使用更短的语法，但现在我们先跳过）。

查询语言是 Javascrip 也有好处，可以像构造对象一样构造操作。这里构造了一个条件，人口必须在 1 万和 100 万之间。

```
var population_range = {}
population_range['$lt'] = 1000000
population_range['$gt'] = 10000
db.towns.find(
  { name : /^P/, population : population_range },
  { name: 1 }
)
{ "_id" : ObjectId("4d0ada87bb30773266f39fe5"), "name" : "Portland" }
```

不限于数字范围，还可以取日期范围。可以找到所有 `last_census` 小于等于 2008 年 1 月 31 日的城镇名称，如下所示：

```
db.towns.find(
  { last_census : { $lte : ISODate('2008-31-01') } },
  { _id : 0, name: 1 }
)

{ "name" : "Punxsutawney" }
```

[1] http://www.pcre.org/

```
{ "name" : "Portland" }
```

请注意我们是怎样阻止输出_id 字段的：显式地将它设置为 0。

5.2.3 深入挖掘

Mongo 喜爱嵌套的数组数据。可以查询精确的匹配值…

```
db.towns.find(
  { famous_for : 'food' },
  { _id : 0, name : 1, famous_for : 1 }
)

{ "name" : "New York", "famous_for" : [ "statue of liberty", "food" ] }
{ "name" : "Portland", "famous_for" : [ "beer", "food" ] }
```

……以及匹配部分值……

```
db.towns.find(
  { famous_for : /statue/ },
  { _id : 0, name : 1, famous_for : 1 }
)

{ "name" : "New York", "famous_for" : [ "statue of liberty", "food" ] }
```

……或按所有匹配的值来查询……

```
db.towns.find(
  { famous_for : { $all : ['food', 'beer'] } },
  { _id : 0, name:1, famous_for:1 }
)

{ "name" : "Portland", "famous_for" : [ "beer", "food" ] }
```

……或排除匹配的值：

```
db.towns.find(
  { famous_for : { $nin : ['food', 'beer'] } },
  { _id : 0, name : 1, famous_for : 1 }
)

{ "name" : "Punxsutawney", "famous_for" : [ "phil the groundhog" ] }
```

但 Mongo 真正的力量来自于深入挖掘文档，并返回深层嵌套子文档的结果。要查询子文档，字段名称就是以点分隔的嵌套层的字符串。

例如，可以找到有独立市长的城镇……

```
db.towns.find(
  { 'mayor.party' : 'I' },
  { _id : 0, name : 1, mayor : 1 }
)

{
  "name" : "New York",
  "mayor" : {
    "name" : "Michael Bloomberg",
    "party" : "I"
  }
}
```

……或那些市长没有注明党派的城镇：

```
db.towns.find(
  { 'mayor.party' : { $exists : false } },
  { _id : 0, name : 1, mayor : 1 }
)

{ "name" : "Punxsutawney", "mayor" : { "name" : "Jim Wehrle" } }
```

如果你想找到匹配单个字段的文档，前面的查询很好，但如果你需要匹配一个子文档的多个字段呢？

1. elemMatch

接下来讨论$elemMatch 指令，从而完成深入挖掘。创建另一个集合，存放国家。这次将利用自选的字符串来覆盖每个_id。

```
db.countries.insert({
  _id : "us",
  name : "United States",
  exports : {
    foods : [
      { name : "bacon", tasty : true },
      { name : "burgers" }
    ]
  }
})
db.countries.insert({
  _id : "ca",
  name : "Canada",
  exports : {
    foods : [
```

```
            { name : "bacon", tasty : false },
            { name : "syrup", tasty : true }
        ]
    }
})
db.countries.insert({
    _id : "mx",
    name : "Mexico",
    exports : {
        foods : [{
            name : "salsa",
            tasty : true,
            condiment : true
        }]
    }
})
```

为了验证加入的国家，可以执行 count 函数，预期的结果是 3。

```
> print( db.countries.count() )
3
```

找出一个国家，它不但出口培根（bacon），而且培根的味道不错。

```
db.countries.find(
    { 'exports.foods.name' : 'bacon', 'exports.foods.tasty' : true },
    { _id : 0, name : 1 }
)

{ "name" : "United States" }
{ "name" : "Canada" }
```

但这不是我们想要的。Mongo 返回了加拿大（Canada），因为它出口培根和味道不错的糖浆（syrup）。这里$elemMatch 就能帮上忙。它规定如果文档（或嵌套的文档）满足所有的条件，该文档就匹配成功。

```
db.countries.find(
    {
        'exports.foods' : {
            $elemMatch : {
                name : 'bacon',
                tasty : true
            }
        }
    },
```

```
    { _id : 0, name : 1 }
)
{ "name" : "United States" }
```

$elemMatch 条件也可以利用高级操作符。可以找出一个国家，它出口味道不错的食品，并且有调味品（condiment）的标签。

```
db.countries.find(
  {
    'exports.foods' : {
      $elemMatch : {
        tasty : true,
        condiment : { $exists : true }
      }
    }
  },
  { _id : 0, name : 1 }
)

{ "name" : "Mexico" }
```

墨西哥（Mexico）正是我们想要的。

2. 布尔操作符

到目前为止，所有的条件都是隐含的“与”操作。如果试图寻找一个国家，它的 name 是 United States，并且_id 是 mx，Mongo 会找不到结果。

```
db.countries.find(
  { _id : "mx", name : "United States" },
  { _id : 1 }
)
```

但是，利用$or 来查找一个或另一个条件，将返回两条结果。这种格式就像前缀表示法：OR A B。

```
db.countries.find(
  {
    $or : [
      { _id : "mx" },
      { name : "United States" }
    ]
  },
  { _id:1 }
)
```

```
{ "_id" : "us" }
{ "_id" : "mx" }
```

操作符很多，这里不能一一介绍，但我们希望这已经让你感受到 MongoDB 的强大查询能力。下面并非全部命令的列表，但包含了很大一部分。

命　令	描　述
$regex	用任意兼容 PCRE 的正则表达式字符串匹配（或就用前面提到的“//”分隔符）
$ne	不等于
$lt	小于
$lte	小于等于
$gt	大于
$gte	大于等于
$exists	检查字段存在
$all	匹配数组中的所有元素
$in	匹配数组中的任一元素
$nin	不匹配数组中的任一元素
$elemMatch	匹配数组或嵌套文档中的所有字段
$or	或
$nour	异或
$size	匹配给定大小的数组
$mod	取余
$type	匹配的定数据类型的字段
$not	对给定操作符检查取反

可以在 MongoDB 的在线文档上找到所有的命令，或从 Mongo 的网站上下载一份备忘表。在接下来的几天里，我们还会再次用到查询。

5.2.4　更新

我们有一个问题。New York 和 Punxsutawney 是够独特的，但添加了 Oregon 的 Portland，或 Maine 的 Portland（或 Texas 或其他）了吗？更新城镇集合，添加美国的一些州。

`update`（条件，操作）函数需要两个参数。第一个参数是条件查询，与传递给 `find()` 的对象一样。第二个参数要么是一个对象（它的字段将取代匹配的文档），要么是一个修改

操作。在这个例子中，修改操作将 `state` 字段设置为字符串 OR。

```
db.towns.update(
  { _id : ObjectId("4d0ada87bb30773266f39fe5") },
  { $set : { "state" : "OR" } }
);
```

你可能会想，为什么需要`$set` 操作呢？Mongo 不是从属性的角度来思考问题的，它只在内部隐含地理解属性，这是为了优化。但接口完全不是面向属性的。Mongo 是面向文档的。你很少需要这样（请注意没有`$set` 操作）：

```
db.towns.update(
  { _id : ObjectId("4d0ada87bb30773266f39fe5") },
  { state : "OR" }
);
```

这会将整个匹配的文档替换成你提供的文档（`{ state : "OR" }`）。既然你没有给出$set 这样的命令，Mongo 就认为你想整个换掉，所以要注意。

可以通过查找来验证更新成功（请注意，使用 `findOne()`仅取出一个匹配对象）。

```
db.towns.findOne({ _id : ObjectId("4d0ada87bb30773266f39fe5") })

{
  "_id" : ObjectId("4d0ada87bb30773266f39fe5"),
  "famous_for" : [
    "beer",
    "food"
  ],
  "last_census" : "Thu Sep 20 2007 00:00:00 GMT-0700 (PDT)",
  "mayor" : {
    "name" : "Sam Adams",
    "party" : "D"
  },
  "name" : "Portland",
  "population" : 582000,
  "state" : "OR"
}
```

可以做的不止是`$set` 一个值。`$inc`（追加一个数）也是很有用的操作。我们让 Portland 的人口增加 1000。

```
db.towns.update(
  { _id : ObjectId("4d0ada87bb30773266f39fe5") },
  { $inc : { population : 1000} }
```

```
)
```

此外还有许多指令，如针对数组的$ positional 操作符。常常会增加新的操作符，在线文档中会更新。下面是主要的指令：

命　令	描　述
$set	将指定字段设置为指定值
$unset	移除该字段
$inc	将指定字段增加指定数值
$pop	从数组中移除最后（或第一个）元素
$push	在数组中添加一个值
$pushAll	在数组中添加所有值
$addToSet	类似 push，但不复制值
$pull	从数组中移除匹配的值
$pullAll	从数组中移除所有匹配的值

5.2.5　引用

前面曾提到，Mongo 不是为了执行联接（join）而设计的。由于它的分布式特点，连接是非常低效的操作。但是，有时候文档相互引用仍然有用。在这种情况下，Mongo 开发团队建议使用{ $ref : "collection_name", $id : "reference_id" }这样的结构。例如，可以更新 towns 集合，让它引用 countries 中的文档。

```
db.towns.update(
  { _id : ObjectId("4d0ada87bb30773266f39fe5") },
  { $set : { country: { $ref: "countries", $id: "us" } } }
)
```

现在可以从 towns 集合中取出 Portland。

```
var portland = db.towns.findOne({ _id : ObjectId("4d0ada87bb30773266f39fe5") })
```

然后，要取出这个城镇的国家，可以用保存的$id 来查询 countries 集合。

```
db.countries.findOne({ _id: portland.country.$id })
```

更好的是，在 JavaScript 中，可以通过引用字段取得 town 文档的集合名称。

```
db[ portland.country.$ref ].findOne({ _id: portland.country.$id })
```

后面两个查询是等价的，第二种更倾向数据驱动。

5.2.6 删除

从集合中删除文档很简单。只要用 remove()来代替 find 函数，所有匹配条件的文档都会删除。要注意，整个匹配的文档都会删除，而不只是匹配的元素或匹配的子文档，这一点很重要。

> **拼字游戏警告**
> 对于拼写错误来说，Mongo 不能很友好地处理。如果你还没遇到这个问题，可能将来会遇到，所以要当心。可以比较静态和动态编程语言。在静态语言中先定义，而动态语言将接受本来不希望的值，甚至是无意思的类型，如 person_name=5。
>
> 文档是没有模式的，所以 Mongo 没法知道你是否希望在城市中插入 population，还是想对 lust_census 查询。它会很开心地插入这样的字段，或告诉你没有匹配的结果。
>
> 灵活性是有代价的。买者自慎之。

在执行 remove()之前，推荐使用 find()来验证条件。在执行操作时，Mongo 不会三思而后行。删除所有出口味道一般的培根的国家。

```
var bad_bacon = {
  'exports.foods' : {
    $elemMatch : {
      name : 'bacon',
      tasty : false
    }
  }
}
db.countries.find( bad_bacon )

{
  "_id" : ObjectId("4d0b7b84bb30773266f39fef"),
  "name" : "Canada",
  "exports" : {
    "foods" : [
      {
```

```
            "name" : "bacon",
            "tasty" : false
          },
          {
            "name" : "syrup",
            "tasty" : true
          }
        ]
      }
    }
```

所有事情看起来都不错。执行删除。

```
db.countries.remove( bad_bacon )
db.countries.count()
2
```

现在执行 count()，就只剩下两个国家了。如果是这样，删除就成功了！

5.2.7　用代码来读取

让我们用一个更有趣的查询选项来结束今天的内容：代码。可以要求 MongoDB 对文档执行一个判断函数。将它放在最后是因为，它应该是最后才想到的手段。这些查询执行得很慢，不能利用索引，Mongo 不能优化它们。但有时候很难拒绝定制代码的力量。

假定要寻找人口在 6000～600 000 的城市。

```
db.towns.find( function() {
  return this.population > 6000 && this.population < 600000;
} )
```

Mongo 甚至为简单的判断函数提供了捷径。

```
db.towns.find("this.population > 6000 && this.population < 600000")
```

可以用$where 子句来执行定制代码的判断条件。在这个例子里，查询还过滤出以 groundhog 闻名的城镇。

```
db.towns.find( {
  $where : "this.population > 6000 && this.population < 600000",
  famous_for : /groundhog/
```

```
})
```

注意：Mongo 将对每个文档以蛮力方式执行这个函数，而我们不能保证给定的字段存在。例如，如果假定 population 字段存在，即使某一个文档中没有 population，整个查询都会失败，因为 JavaScript 不能够正确地执行。写定制的 JavaScript 函数时要当心，在尝试定制代码时要先熟悉 JavaScript 的使用。

5.2.8　第 1 天总结

今天，我们接触了我们的第一个文档数据库，MongoDB。我们看到如何将嵌套的数据另存为 JSON 对象，并查询任何深度的数据。你知道，文档数据库可以看成是关系模型中的一行数据，但没有模式，以生成的`_id`为主键。在 Mongo 中，一组文档称为一个集合，类似于 PostgreSQL 中的表。

我们以前遇到的方式是存储一些简单数据类型的集合，Mongo 与此不同，保存复杂的、非标准化的文档，存取任意 JSON 结构的集合。Mongo 在这种灵活的存储策略上提供了强大的查询机制，这是任何预定义的模式都不能提供的。

由于非标准化的特点，文档存储很适合存放品质未知的数据，而其他的方式（如关系型和列型）希望你事先知道，添加或修改字段时需要改变模式。

第 1 天作业

求索

1．将 MongoDB 的在线文档加入书签。

2．查看如何在 Mongo 中创建正则表达式。

3．熟悉命令行`db.help()`和`db.collections.help()`的输出。

4．找到你使用的语言（Ruby、Java、PHP 等）的 Mongo 驱动程序。

实践

1．输出一份 JSON 文档，它包含`{ "hello" : "world" }`。

2．利用不区分大小写的正则表达式，找出包含单词 new 的城镇。

3．找到所有名称中包含字母 e，并且以 food 或 beer 闻名的城镇。

4．创建一个名为 blogger 的新数据库，其中包含一个名为 articles 的集合，插入一篇

新文章，包含作者名称、email、创建日期和文字内容。

5．更新这篇文章，加入一个评论数组，评论包含作者和文字内容。

6．执行外部 JavaScript 文件中包含的查询。

5.3　第 2 天：索引、分组和 mapreduce

提高 MongoDB 的查询性能是今天的第一项学习内容，接下来是一些更强大、更复杂的分组查询。最后，利用 mapreduce 来进行一些分析，像我们在 Riak 中做的那样，从而完成今天的学习。

5.3.1　索引：如果还不够快

Mongo 有一个有用的内置特征，就是用索引来增加查询的速度，我们知道，不是所有 NoSQL 数据库都有这个特征。MongoDB 采用了一些最好的数据结构实现索引，如经典的 B 树，以及其他一些附加方法，如两维和球形地理空间（shperical GeoSpatial）索引。

接下来要做一个小实验，看看 MongoDB 的 B 树索引的力量。将加入一些电话号码，它们的国家码是随机选取的（可以自由地用自己的国家码来替换）。在控制台输入下面的代码。它将生成 100 000 个电话号码（可能要一点时间），范围介于 1-800-555-0000 到 1-800-565-9999。

```
mongo/populate_phones.js
populatePhones = function(area,start,stop) {
  for(var i=start; i < stop; i++) {
    var country = 1 + ((Math.random() * 8) << 0);
    var num = (country * 1e10) + (area * 1e7) + i;
    db.phones.insert({
      _id: num,
      components: {
        country: country,
        area: area,
        prefix: (i * 1e-4) << 0,
        number: i
      },
      display: "+" + country + " " + area + "-" + i
    });
  }
```

```
}
```

执行这个函数，参数是 3 位的地区码（如 800），以及一个 7 位的号码范围（5 550 000～5 650 000，输入时请检查 0 的个数）。

```
populatePhones( 800, 5550000, 5650000 )
db.phones.find().limit(2)

{ "_id" : 18005550000, "components" : { "country" : 1, "area" : 800,
  "prefix" : 555, "number" : 5550000 }, "display" : "+1 800-5550000" }
{ "_id" : 88005550001, "components" : { "country" : 8, "area" : 800,
  "prefix" : 555, "number" : 5550001 }, "display" : "+8 800-5550001" }
```

在创建任何一个新集合时，Mongo 会自动在_id 上建立索引。这些索引可以在 system.indexes 中看到。下面的查询展示了数据库中所有的索引：

```
db.system.indexes.find()
{ "name" : "_id_", "ns" : "book.phones", "key" : { "_id" : 1 } }
```

因为大多数查询不只涉及_id，所以需要在这些字段上建立索引。

将在 display 字段上创建一个 B 树索引。但首先，验证索引将改善速度。要做到这一点，先在没有索引时执行一个查询。explain()方法用于输出给定操作的细节。

```
db.phones.find({display: "+1 800-5650001"}).explain()

{
  "cursor" : "BasicCursor",
  "nscanned" : 109999,
  "nscannedObjects" : 109999,
  "n" : 1,
  "millis" : 52,
  "indexBounds" : {
  }
}
```

输出会与我们的不一样，但请注意 millis 字段（完成查询的毫秒数），它可能是两位数。

通过调用 ensureIndex(fields,options)，在集合上创建索引。参数 fields 是一个对象，包含要建索引的那些字段。参数 options 描述了要建的索引类型。在这个例子里，将在 display 上建立唯一索引，它将丢弃重复的条目。

```
db.phones.ensureIndex(
  { display : 1 },
```

```
  { unique : true, dropDups : true }
)
```

现在重试 find()，检查 explain()，看看情况是否改善。

```
db.phones.find({ display: "+1 800-5650001" }).explain()

{
  "cursor" : "BtreeCursor display_1",
  "nscanned" : 1,
  "nscannedObjects" : 1,
  "n" : 1,
  "millis" : 0,
  "indexBounds" : {
    "display" : [
      [
        "+1 800-5650001",
        "+1 800-5650001"
      ]
    ]
  }
}
```

millis 的值从 52 降到了 0，改进是无穷大（52/0）！这是开玩笑，但速度的提高幅度是数量级式的。同时也注意，cursor（光标）从 Basic 变为 B 树（称为光标是因为它指向存放的值，它不包含这些值）。Mongo 不再扫描整个集合，而是通过查找树来取得值。重要的是，扫描的对象从 109 999 降到了 1，因为它变成了在唯一索引上的查找。

explain()是一个有用的函数，但只在测试特定的查询调用时才会用它。如果需要在普通的测试或生产环境中进行分析，就需要系统剖析器（System Profiler）。

将剖析级别设置为 2（级别 2 将保存所有的查询，级别 1 只保存超过 100 毫秒的、比较慢的查询），然后像往常一样运动 fird()。

```
db.setProfilingLevel(2)
db.phones.find({ display : "+1 800-5650001" })
```

这将在 system.profile 集合中创建一个新对象，可以像读其他表一样读取它。ts 是执行查询时的时间戳，info 是操作的字符串描述，millis 是它所花的时长。

```
db.system.profile.find()

{
  "ts" : ISODate("2011-12-05T19:26:40.310Z"),
```

```
    "op" : "query",
    "ns" : "book.phones",
    "query" : { "display" : "+1 800-5650001" },
    "responseLength" : 146,
    "millis" : 0,
    "client" : "127.0.0.1",
    "user" : ""
}
```

像昨天的嵌套查询一样，Mongo 可以在嵌套的值上建立索引。如果你希望在所有的地区码上建立索引，使用带点的字段表示法：`components.area`。在产品环境中，应该总是使用`{ background : 1 }`选项，在后台建立索引。

```
db.phones.ensureIndex({ "components.area": 1 }, { background : 1 })
```

如果利用 `find()`发现所有的 `phones` 集合上的系统索引，新的应该出现在最后。第一个索引总是自动创建的，以便按`_id` 快速查询，第二个是前面建立的唯一索引。

```
db.system.indexes.find({ "ns" : "book.phones" })

{
    "name" : "_id_",
    "ns" : "book.phones",
    "key" : { "_id" : 1 }
}
{
    "_id" : ObjectId("4d2c96d1df18c2494fa3061c"),
    "ns" : "book.phones",
    "key" : { "display" : 1 },
    "name" : "display_1",
    "unique" : true,
    "dropDups" : true
}
{
    "_id" : ObjectId("4d2c982bdf18c2494fa3061d"),
    "ns" : "book.phones",
    "key" : { "components.area" : 1 },
    "name" : "components.area_1"
}
```

在 book.phones 上的索引漂亮地完成了。

在本节结束时，我们应该注意，在大型的集合上创建索引可能较慢，并且占用较多的

资源。在建立索引时，总是应该考虑这些冲击，不要在高峰时间进行，要在后台进行，要手动进行，而不是自动创建索引。网上还有更多索引技巧，但这些是一些基本技巧，最好知道。

5.3.2 聚合查询

昨天研究的查询，对于基本的数据抽取是有用的，但后面的处理需要你来解决。例如，假定我们想统计大于 559–9999 的电话号码个数，我们希望数据库在后面执行这样的计数。像 PostgreSQL 一样，count()是最基本的聚合器。它进行查询，并返回匹配文档的个数。

```
db.phones.count({'components.number': { $gt : 5599999 } })

50000
```

> **改变是好的**
> 聚合查询返回的结构与我们习惯的单个文档不同。count()将结果聚合成文档的计数，distinct()将结果聚合成一个数组，group()返回的文档是用它自己的设计。即使是 mapreduce，通常也要花一些努力，取出类似于内部存储文档的对象。

为了体现接下来几个聚合查询的威力，再向 phones 集合中添加 100 000 个电话号码，这次使用另一个地区码。

```
populatePhones( 855, 5550000, 5650000 )
```

如果多个文档匹配条件，distinct()命令返回每个匹配的值（不是整个文档）。可以像下面这样取得小于 5 550 005 的不同 components.number：

```
db.phones.distinct('components.number', {'components.number': { $lt : 5550005 } })

[ 5550000,5550001,5550002,5550003,5550004 ]
```

虽然有两个 5 550 005（一个地区码是 800，另一个是 855），但在结果中只出现一次。

group()聚合查询类似 SQL 中的 GROUP BY。它也是 Mongo 中最复杂的基本查询。可以对大于 5599999 的电话号码计数，并且按地区码将结果放入不同的分组。其中 key 是按其分组的字段，cond（条件）是我们感兴趣的值的范围，reduce 是一个函数，管理如何输出值。

还记得第 3 章中的 mapreduce 吗？数据已经映射到原有的文档集合。不再需要映射了，只要对文档进行归约。

```
db.phones.group({
  initial: { count:0 },
  reduce: function(phone, output) { output.count++; },
  cond: { 'components.number': { $gt : 5599999 } },
  key: { 'components.area' : true }
})

[ { "components,area":"800" : "leunt": 50000, "lomponents,area": "855" : "count": 50000 } ]
```

诚然，接下来的两个例子是奇怪的使用场景。它们只是用来展示 group() 的灵活性。

可以用下面的 group() 调用来重现 count() 函数的结果。这里省去了聚合键：

```
db.phones.group({
  initial: { count:0 },
  reduce:  function(phone, output) { output.count++; },
  cond:    { 'components.number': { $gt : 5599999 } }
})

[ { "count" : 100000 } ]
```

这里首先建立一个初始对象，它包含一个 count 字段，初值是 0（这里创建的字段将出现在输出中）。接下来描述对这个字段做什么，即声明一个归约函数，对遇到的每个文档，让计数加 1。最后，给出了一个分组条件，限制哪些文档需要归约。结果与 count() 是一样的，因为条件是一样的。这里省去了键，因为希望所有匹配的文档都加入结果集。

也可以重现 distinct() 函数。出于性能的考虑，先创建一个对象，将一些数字作另存为字段（实际上创建了一个自由定义的集合）。在规约函数中（它针对每个匹配的文档执行），将值设置为 1，作为占位符（它是我们想要的字段）。

从技术上这就是所需要的。但是，如果我们真想重现 distinct()，就应该返回一个整数数组。所以添加了一个 finalize(out) 方法，它在返回值之前的最后一刻执行，将对象转换成一个字段值数组。这个函数接着将这些字符串转换成整数（如果你希望看到事情完成的过程，可以去掉 finalize 函数集，执行下面的命令）。

```
db.phones.group({
  initial: { prefixes : {} },
  reduce: function(phone, output) {
    output.prefixes[phone.components.prefix] = 1;
  },
  finalize: function(out) {
    var ary = [];
```

```
        for(var p in out.prefixes) { ary.push( parseInt( p ) ); }
        out.prefixes = ary;
    }
})[0].prefixes

[ 555,556,557,558,559,560,561,562,563,564 ]
```

group()函数很强大（像 SQL 的 GROUP BY 一样），但 Mongo 的实现也有不利的一面。首先，结果不能超过 10 000 个文档。而且，如果你对 Mongo 集合进行了分片（明天将讨论），group()就无效了。还有很多灵活的方式来创建查询。出于这样或那样的原因，稍后将深入 MongoDB 的 mapreduce 实现。但首先，我们将弄清楚客户端命令和服务器端命令的区别，它对应用有重要影响。

5.3.3　服务器端命令

如果试图通过命令行执行下面的函数（或通过驱动程序），客户端会将电话文档逐个取到本地（所有 100 000 个），然后再将每个电话文档逐个保存到服务器上。

```
mongo/update_area.js
update_area = function() {
    db.phones.find().forEach(
        function(phone) {
            phone.components.area++;
            phone.display = "+"+
                phone.components.country+" "+
                phone.components.area+"-"+
                phone.components.number;
            db.phones.update({ _id : phone._id }, phone, false);
        }
    )
}
```

但是，Mongo 的 db 对象提供了一个名为 eval()的命令，它将指定的函数传给服务器。这极大地减少了客户端与服务器之间的对话，因为代码是在远端执行的。

```
> db.eval(update_area)
```

除了执行 JavaScript 函数，Mongo 中还有几个预设的命令，其中大部分是在服务器端执行的，虽然有些要求只能在 admin 数据库上执行（可以通过 use admin 来访问）。

```
> use admin
> db.runCommand("top")
```

top 命令将输出服务器上所有集合的访问细节。

```
> use book
> db.listCommands()
```

在执行 listCommands()时，你可能会注意到我们用过的许多命令。实际上，可以通过 runCommand()方法执行许多常见命令，如统计电话的个数。但是，你可能会注意到输出有一点区别。

```
> db.runCommand({ "count" : "phones" })
{ "n" : 100000, "ok" : 1 }
```

返回的个数 *n* 是正确的（100 000），但格式是一个对象，包含 ok 字段。这是因为 db.phones.count()是一个封装函数，为了方便，通过 shell 的 JavaScript 接口来创建的，而 runCommand()是在服务器上执行计数。我们曾提到过，可以不带圆括号调用，查看像 count()这样的函数是如何工作的。

```
> db.phones.count
function (x) {
    return this.find(x).count();
}
```

有趣！collection.count()只是一种方便的封装，实际上先调用 find()（它本身又是一层封装，实际上是一个本地查询对象，返回指向结果集的光标），再对 find()的结果调用 count()。

```
> db.phones.find().count
```

你会看到一个大得多的函数（大到不方便在这里列出）。但请看一下，在一些设置代码之后，你会看到下面这样的代码：

```
var res = this._db.runCommand(cmd);
if (res && res.n != null) {
    return res.n;
}
```

有趣加倍了！count()执行了 runCommand()，并返回字段 *n* 的值。

1. runCommand

在深入研究方法的工作原理时，让我们来看看 runCommand()函数。

```
> db.runCommand
function (obj) {
    if (typeof obj == "string") {
        var n = {};
        n[obj] = 1;
        obj = n;
    }
    return this.getCollection("$cmd").findOne(obj);
}
```

结果表明，runCommand()也是一个辅助函数，封装了一个调用，该调用针对名为$cmd 的集合。可以直接对这个集合调用，从而执行任何方法。

```
> db.$cmd.findOne({'count' : 'phones'})
{ "n" : 100000, "ok" : 1 }
```

这就是最底层的机制了，也是驱动程序与 Mongo 服务器通信的一般方式。

2. 岔开话题

这里岔开话题有两个理由：

- 彻底说清楚 mongo 控制台上执行的大多数命令，实际上是在服务器上执行的，而不是在客户端上执行。客户端只是提供了一些方便的封装函数。
- 可以利用执行服务器端代码的概念，实现我们自己的一些目的，在 MongoDB 中创建类似 PostgreSQL 中存储过程的对象。

任何 JavaScript 函数都可以保存在一个名为 system.js 的特殊集合中。这是一个普通集合，可以将函数名作为_id，函数体作为 value。

```
> db.system.js.save({
    _id:'getLast',
    value:function(collection){
      return collection.find({}).sort({'_id':1}).limit(1)[0]
    }
})
```

接下来，通常会直接在服务器上执行它。eval()函数将这个字符串传给服务器，将

它作为 JavaScript 代码来执行，并返回结果。

```
> db.eval('getLast(db.phones)')
```

它返回的值应该与在本地调用 `getLast(collection)` 一样。

```
> db.system.js.findOne({'_id': 'getLast'}).value(db.phones)
```

值得一提的是，`eval()` 在执行时会阻塞 `mongod`，所以它主要适用于快速执行的命令和测试，而不是常见的产品过程。也可以在`$where` 和 mapreduce 中使用这个函数。武器库还有最后一样武器，即在 MongoDB 中开始执行 mapreduce。

5.3.4 mapreduce（以及Finalize）

Mongo 的 mapreduce 方式与 Riak 的类似，有一些小区别。不是由 map()函数返回转换过的值，而是要求映射函数（mapper）用一个主键去调用 `emit()` 函数。这样做的好处在于，可以让每个文档发出不止一次。`reduce()` 函数接受单个键和发送给该键的一个值列表。最后 Mongo 提供了可选的第 3 步，名为 `finalize()`，它在规约函数（reducer）执行之后，对每个映射的值仅执行一次。这样你就能够执行最终的计算，或完成需要的清理工作。

我们已经清楚地知道了 mapreduce 的基础知识，接下来将快速越过戏水池，直接进入深水区。生成一份报表，对每个国家中包含同样数字的电话号码进行计数。首先，要保存一个辅助函数。它抽取所有不同的数字，构成一个数组（要理解整个 mapreduce，并不需要理解这个辅助函数的工作原理）。

```
mongo/distinct_digits.js
distinctDigits = function(phone){
  var
    number = phone.components.number + '',
    seen = [],
    result = [],
    i = number.length;
  while(i--) {
    seen[+number[i]] = 1;
  }
  for (i=0; i<10; i++) {
    if (seen[i]) {
      result[result.length] = i;
    }
```

```
    }
    return result;
}
db.system.js.save({_id: 'distinctDigits', value: distinctDigits})
```

在 mongo 命令行载入这个文件。如果该文件在启动 mongo 的同一个目录下，只需要文件名就可以了；否则，需要完整的路径。

```
> load('distinct_digits.js')
```

完成之后，可以做一个快速测试（如果你遇到麻烦，别介意添加一些简单的 print() 函数）。

```
db.eval("distinctDigits(db.phones.findOne({ 'components.number' : 5551213 }))")
[ 1,2,3,5 ]
```

现在，可以着手映射函数了。对于任何 mapreduce 函数来说，决定哪些字段要映射是关键，因为这规定了返回的聚合值。因为报表负责找出不同的数字，不同值的数组是一个字段。但由于需要按 country（国家）来查询，因此还有另一个字段。将这两个值作为一个复合键：{digits : X, country : Y}.

因为目标只是对这些值计数，所以发出值 1（每个文档代表了要计数的一项）。规约函数的任务是将所有这些 1 加在一起。

```
mongo/map_1.js
map = function() {
    var digits = distinctDigits(this);
    emit({digits : digits, country : this.components.country}, {count : 1});
}
```

```
mongo/reduce_1.js
reduce = function(key, values) {
    var total = 0;
    for(var i=0; i<values.length; i++) {
        total += values[i].count;
    }
    return { count : total };
}

results = db.runCommand({
    mapReduce: 'phones',
```

```
  map: map,
  reduce: reduce,
  out: 'phones.report'
})
```

因为通过 out 参数设置了集合的名称（out : 'phones.report'），所以可以像对其他集合那样查询结果。它是一个物化视图（materialized view），可以在查看表清单时看到。

```
> db.phones.report.find({'_id.country' : 8})
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 4, 5, 6 ], "country" : 8 },
  "value" : { "count" : 19 }
}
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 5 ], "country" : 8 },
  "value" : { "count" : 3 }
}
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 5, 6 ], "country" : 8 },
  "value" : { "count" : 48 }
}
{
  "_id" : { "digits" : [ 0, 1, 2, 3, 5, 6, 7 ], "country" : 8 },
  "value" : { "count" : 12 }
}
还有更多
```

输入 it，以便继续遍历结果集。请注意，唯一发出的键在_id 字段中，所有从规约函数返回的数据在字段 value 中。

如果你希望这个 mapreduce 函数只输出结果，而不是将结果放到一个集合中，可以将 out 值设置为{ inline : 1 }，但要记住，输出的结果是有大小限制的。对于 Mongo 2.0 来说，这个限制是 16MB。

还记得在第 3 章中，规约函数可以既不用映射的（发出的）结果，也不用其他规约函数的结果作为输入。如果两个规约函数映射到相同的主键上，为什么一个规约函数的输出要作为另一个的输入？请考虑一下，如果运行在不同的服务器上，将是怎样的情形。图 5-3 展示了这种情况。

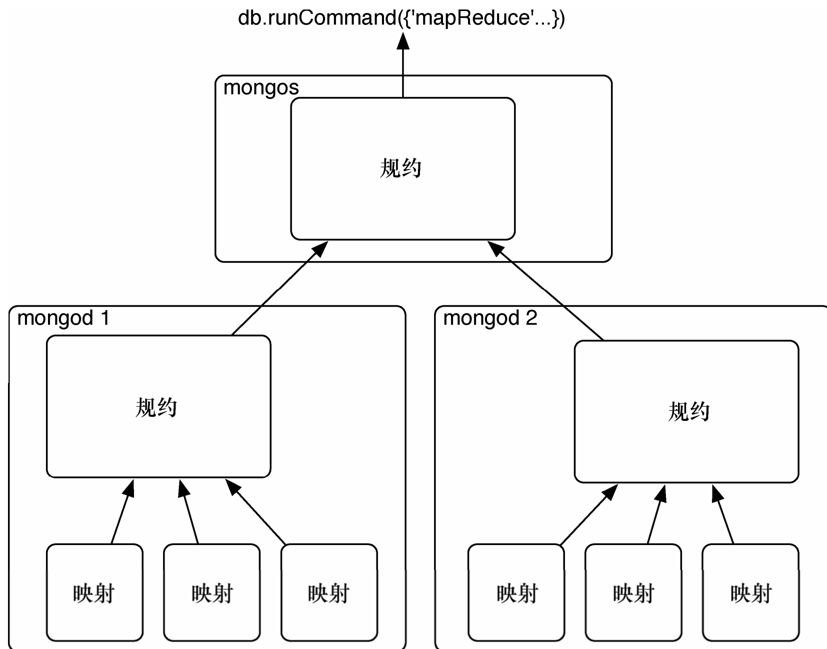


图 5-3 Mongo 在两个服务器上调用映射和规约

每个服务器必须执行它自己的 map() 和 reduce() 函数，然后将这些需要合并的结果推给最初发起调用的服务，将它们收集起来。经典的分而治之。如果将规约函数的输出重命名为 total（总和），而不是 count（计数），就需要在循环中同时处理两种情况，如下所示：

```
mongo/reduce_2.js
reduce = function(key, values) {
  var total = 0;
  for(var i=0; i<values.length; i++) {
    var data = values[i];
    if('total' in data) {
      total += data.total;
    } else {
      total += data.count;
    }
  }
  return { total : total };
}
```

但是，Mongo 预计你可能需要执行某些最后的修改，诸如，重命名一个字段，或进行一些其他计算。如果确实需要输出字段是 total，那么可以实现一个 finalize() 函数，

它的工作方式与 `group()` 中的 `finalize` 函数是一样的。

5.3.5　第 2 天总结

第 2 天我们通过引入一些聚合查询，增强了查询的威力：`count()`、`distinct()`，最后是 `group()`。为了减少这些查询的响应时间，使用了 MongoDB 的索引选项。如果需要更强大，始终存在的 `mapReduce()` 是可用的。

第 2 天作业

求索

1．执行管理（admin）命令的快捷方式。

2．查询和光标的在线文档。

3．mapreduce 的 MongoDB 文档。

4．通过 JavaScript 接口，研究 3 个集合函数的代码：`help()`、`findOne()` 和 `stats()`。

实践

1．实现一个 finalize 方法，将 count 作为 total 输出。

2．安装一种选定语言的 Mongo 驱动程序，并连接到数据库。通过它填充一个集合，并在其中一个字段上建立索引。

5.4　第 3 天：副本集、分片、地理空间和 GridFS

Mongo 具有强大的能力，可以用不同的方式来存储和查询数据。但是其他数据同样也能做到。文档数据库的独特之处在于，它们能够有效地处理任意嵌套的、无模式的数据文档。Mongo 在文档存储领域的独特之处在于，它能够在多台服务器上实现伸缩，对集合进行复制（复制数据到其他服务器）或分片（将一个集合分成几片），并行执行查询。复制和分片都提高了可用性。

5.4.1　副本集

Mongo 的设计目的是能够伸缩，而不是单独运行。它是为数据一致性和分区容错性而生的，但数据分片是有成本的：如果一部分丢失，就会危及整个集合。查询只包含西半球

国家的集合，有什么意义呢？对这种分片的隐含弱点，Mongo 处理的方法很简单：复制。图 5-4 说明了复制的重要性。你几乎不应该在生产环境中使用单个 Mongo 实例，而是应该在多个服务器上复制存储的数据。



图 5-4　复制的重要性

今天，我们不再摆弄原有的数据库，而是从头开始建立一些新服务器。Mongo 的默认端口是 27017，所以将在其他端口启动服务器。前面曾提到，必须先创建数据目录，所以创建 3 个目录：

```
$ mkdir ./mongo1 ./mongo2 ./mongo3
```

接下来启动 Mongo 服务器。这次加上 `replSet` 标记，取名为 book，并指定端口。在做这件事时，打开 REST 标记，这样就能用 `Web` 接口。

```
$ mongod --replSet book --dbpath ./mongo1 --port 27011 --rest
```

打开另一个终端窗口，执行下一条命令，启动另一个服务器，指向不同的目录，监听另一个端口。然后打开第 3 个终端窗口，启动第 3 个服务器。

```
$ mongod --replSet book --dbpath ./mongo2 --port 27012 --rest
$ mongod --replSet book --dbpath ./mongo3 --port 27013 --rest
```

请注意，你会在输出中看到很多这样的信息。

```
[startReplSets] replSet can't get local.system.replset config from self \
or any seed (EMPTYCONFIG)
```

这是好事，我们还没有初始化副本集，Mongo 向我们告知。启动一个 `mongo shell`，连接一个服务器，执行 `rs.initiate()` 函数。

```
$ mongo localhost:27011
> rs.initiate({
  _id: 'book',
  members: [
    {_id: 1, host: 'localhost:27011'},
    {_id: 2, host: 'localhost:27012'},
    {_id: 3, host: 'localhost:27013'}
  ]
})
> rs.status()
```

注意，用到了新名对象 rs（副本集）。像其他对象一样，它也有 help()方法供调用。执行 status()命令，可以知道何时副本集在执行，所以在继续操作之前，要不断检查完成的状态。如果查看 3 个服务器的输出，应该看到一个服务器输出下面一行内容：

```
[rs Manager] replSet PRIMARY
```

另两个服务器将输出：

```
[rs_sync] replSet SECONDARY
```

PRIMARY 将作为主服务器。很可能这是监咱 27011 端口的服务器（因为它先启动），但如果不是这样，只要再启动一个控制台，连接主服务器。在命令行中插入任意老数据，并且要做一个实验。

```
> db.echo.insert({ say : 'HELLO!' })
```

插入之后，退出控制台，然后要关闭主节点，测试变更已经复制。按 CTRL+C 快捷键就够了。如果查看剩下两个服务器的日志，应该看到其中一个已经升级成为主服务器（它将输出 replSet PRIMARY）。打开控制台并连接它（对我们来说，它是 localhost:27012），db.echo.find()应该包含插入的值。

我们还要玩一个“控制台推诿”的游戏。打开一个控制台，连接剩下的 SECONDARY（从）服务器。为了确认，执行 isMaster()函数。结果是这样的：

```
$ mongo localhost:27013
MongoDB shell version: 1.6.2
connecting to: localhost:27013/test
> db.isMaster()
{
  "setName" : "book",
  "ismaster" : false,
  "secondary" : true,
```

```
  "hosts" : [
    "localhost:27013",
    "localhost:27012",
    "localhost:27011"
  ],
  "primary" : "localhost:27012",
  "ok" : 1
}
```

在这个 shell 中，尝试插入另一个值。

```
> db.echo.insert({ say : 'is this thing on?' })
not master
```

消息 not master 告诉我们，我们不能写入从结点。你也不能直接从它读取。每个副本集只能有一个主节点，你必须与它打交道。它是这个副本集的入口。

与单一数据源的数据库相比，复制数据有它自己的问题。在 Mongo 设置中，要决定主节点宕机时谁提升为主节点，这是一个问题。Mongo 处理这个问题的办法，是让每个 mongod 服务有一张选票，拥有最新数据的服务将提升为主节点。现在应该还有两个 mongod 服务在运行。继续关掉当前的主节点。记住，当对 3 个节点这样做时，另两个节点中的一个会提升为新的主节点。但这次情况有所不同。最后剩下的服务器的输出会像这样：

```
[ReplSetHealthPollTask] replSet info localhost:27012 is now down (or...
[rs Manager] replSet can't see a majority, will not try to elect self
```

这就要归结到 Mongo 建立服务器的哲学，以及为什么总是应该有奇数个服务器（3 个、5 个等）。

接下来重新启动另两个服务器，并查看日志。当两个节点重新出现时，它们就进入恢复状态，并试图与新的主节点再次同步数据。“怎么这么倒霉！？”（我们听到你在叫）。“那么，如果原来的主节点还有一些数据没有传播出去，怎么办？”这些操作放弃了。直到大多数节点都有了数据的副本时，对 Mongo 副本集的写入才认为成功。

5.4.2　偶数节点的问题

复制的概念很容易接受：写入一个 MongoDB 服务器，把数据复制到该副本集的其他服务器上。如果一个服务器不可用，其他服务器中的一个会提升为主节点，并响应请求。但除了服务器崩溃之外，还有其他方式导致服务器不可用。有时候，节点之

间的网络连接断了。在这种情况下 Mongo 规定，如果大部分节点还能通信，就认为网络还有效。

MongoDB 希望副本集中总节点数为奇数。例如，考虑 5 个节点的网络。如果两连接问题导致它分裂成 2 节点和 3 节点的片段，较大的片段明显占了多数，可以选出一个主节点，继续响应请求。如果没有明显的多数，就无法选出。

为了说明为什么希望是奇数节点，请考虑 4 节点的副本集会发生什么情况。假定网络分区导致两台服务器与另两台服务器失去联系。一边有最初的主节点，但因为没有明显的大多数，主节点停止服务。另一边也类似，不能选出主节点，因为它也不能与明显的大多数节点通信。两边现在都不能处理请求，系统实际上就宕机了。奇数个节点发生这种情况的可能性比较小，即分段的网络导致每个片段都少于明显的大多数。

有些数据库（如 CouchDB）的设计支持多个主节点，但 Mongo 不是这样，所以它没有准备在多个主节点之间提供数据更新。MongoDB 解决多主节点冲突的方法很简单，它不允许有多个主节点。

不像 Riak，Mongo 总是知道最新的值，客户端不需要决定。Mongo 关心很强的写入一致性，为了实现这一点，防止出现多主节点是不错的方法。

5.4.3 分片

Mongo 存在的一个核心理由，就是安全而快速地处理非常大的数据集。实现这个目标最清楚的方法，就是按值的范围进行横向分片，或简称*分片*（sharding）。不同于一个服务器存放一个集合的所有值，把有些范围的值切分（或者说分片）到其他服务器上。例如，在电话号码集合中，可以将所有小于 1-500-000-0000 的电话号码放到 Mongo 服务器 A 上，将大于等于 1-500-000-0001 的放到服务器 B 上。Mongo 通过自动分片、自动管理这种划分，从而使这样做变得更容易。

启动一对非复制的 mongod 服务器。像副本集一样，规划的分片服务器需要特殊的参数（它只是表明这个服务器能够分片）。

```
$ mkdir ./mongo4 ./mongo5
$ mongod --shardsvr --dbpath ./mongo4 --port 27014
$ mongod --shardsvr --dbpath ./mongo5 --port 27015
```

现在需要一个服务器，实际追踪键。假定创建了一个表，按字母顺序保存城市的名称。需要某种方式知道（举个例子）以 A～N 开头的城市放在服务器 mongo4 上，以 O～Z 开头的城市放在服务器 mongo5 上。在 Mongo 中，创建一个 config（配置）服务器（它也是

一个常规的 mongod），追踪哪个服务器（mongo4 或 mongo5）拥有哪些值。

```
$ mkdir ./mongoconfig
$ mongod --configsvr --dbpath ./mongoconfig --port 27016
```

最后，需要运行第 4 个服务器，名为 mongos，它是对客户的一个单点入口。mongos 服务器将连接到 mongoconfig 配置服务器，追踪存放在上面的分片信息。设置端口为 27020，chunkSize 是 1。（chunkSize 是 1MB，是允许的最小值。这只是针对小数据集，这样我们可以看到分片发生。在生产环境中，应该使用默认值，或更大的数字。）通过 -configdb 标识，让 mongos 指向配置服务器和端口。

```
$ mongos --configdb localhost:27016 --chunkSize 1 --port 27020
```

mongos 漂亮的地方在于，它是一个全功能 mongod 服务器的轻量级副本。对 mongod 发出的几乎所有命令，都可以对 mongos 发出，这使它成为了客户端与多个分片服务器之间的完美中介。服务器设置图也许有助于理解（见图 5-5）。

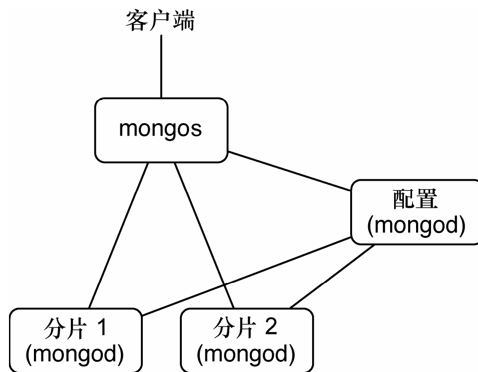


图 5-5　小型分片集群

> **投票与仲裁者**
>
> 有时候你可能不希望用奇数台服务器来复制数据。此时，要么可以启动一个仲裁者（一般推荐），要么增加服务器的投票权重（一般不推荐）。在 Mongo 中，仲裁者（arbiter）是副本集中参与投票，但不复制的服务器。像启动其他服务器一样启动它，但在配置中设一个标识，像这样：{_id: 3, host: 'localhost:27013', arbiterOnly : true}。仲裁者对于连接中断的情况是有用的，就像美国副总统在参议院一样。默认情况下，每个 mongod 实例有一张选票。

> **mongos 与 mongoconfig**
>
> 你可能好奇，为什么 Mongo 将 configuration 和 mongos 入口点分别放在两个服务器上。这是因为在生产环境中，它们通常会在不同的物理服务器上。配置服务器（它自己被复制）为其他分片服务器管理分片信息，而 mongos 可能只存在于本地的应用服务器上，让客户端能够方便地连接（不需要管理连接哪个分片）。

现在打开 mongos 服务器的控制台，进入 admin 数据库。要配置一些分片。

```
$ mongo localhost:27020/admin
> db.runCommand( { addshard : "localhost:27014" } )
{ "shardAdded" : "shard0000", "ok" : 1 }
> db.runCommand( { addshard : "localhost:27015" } )
{ "shardAdded" : "shard0001", "ok" : 1 }
```

这样设置之后，就要给出需要分片的数据库和集合，以及分片所依据的字段（在例子中，是城市的名称）。

```
> db.runCommand( { enablesharding : "test" } )
{ "ok" : 1 }
> db.runCommand( { shardcollection : "test.cities", key : {name : 1} } )
{ "collectionsharded" : "test.cities", "ok" : 1 }
```

完成所有设置之后，加载一些数据。如果下载了本书的代码，就会看到一个 12MB 的数据文件，名为 mongo_cities1000.json，它包含世界上所有人口超过 1000 的城市。下载这个文件，执行下面的导入脚本，将数据导入 mongos 服务器：

```
$ mongoimport -h localhost:27020 -db test --collection cities \
  --type json mongo_cities1000.json
```

通过 mongos 控制台，输入 use test，从 admin 环境回到 test 环境。

5.4.4 地理空间查询

Mongo 有一个内置的漂亮技巧。虽然我们今天关注的是服务器的设置，但是每天都应该有一点妙招，这就是 Mongo 快速执行地理空间查询的能力。先连接到 mongos 分片服务器。

```
$ mongo localhost:27020
```

地理空间查询的秘诀就在于索引。这是一种特殊形式的索引地理数据，名为 geohash，不仅能在任意的查询中快速找到具体的值或范围，而且能快速找到附近的值。方便的是，

在前一节的末尾，装入了很多地理数据。所以要查询它，第 1 步就是在 location 字段上对数据进行索引。2 维索引必须建立在两个值的字段上，在例子中是一个哈希表（例如，{ longitude:1.48453，latitude:42.57205 }），但它也很容易是一个数组（例如，[1.48453，42.57205]）。

```
> db.cities.ensureIndex({ location : "2d" })
```

如果不是处理分片的集合，就可以很容易地查询城市或附近的位置。但是，对当前的 Mongo 版本来说，下面的查询只能在不分片的集合上生效。

```
> db.cities.find({ location : { $near : [45.52, -122.67] } }).limit(5)
```

在将来的版本中，应该提供分片集合的补丁。但在目前，要对分片的 cities 集合进行查询，找出一个位置附近的其他城市，可以使用 geoNear() 命令。下面是它返回结果的一个例子：

```
> db.runCommand({geoNear : 'cities', near : [45.52, -122.67],
  num : 5, maxDistance : 1})
{
  "ns" : "test.cities",
  "near" : "1000110001000000011100101011100011001001110001111110",
  "results" : [
    {
      "dis" : 0.007105400003747849,
      "obj" : {
        "_id" : ObjectId("4d81c216a5d037634ca98df6"),
        "name" : "Portland",
        ...
      }
    },
    ...
  ],
  "stats" : {
    "time" : 0,
    "btreelocs" : 53,
    "nscanned" : 49,
    "objectsLoaded" : 6,
    "avgDistance" : 0.02166813996454613,
    "maxDistance" : 0.07991909980773926
  },
  "ok" : 1
}
```

geoNear()也有助于发现一些地理空间命令的错误。它返回的有用信息是一座金矿，诸如与查询点的距离，返回集的平均距离和最大距离，以及索引信息。

5.4.5 GridFS

分布式系统有一点不足，就是缺乏一个一致的文件系统。假定你运营着一个网站，用户可以上传他们自己的图片。如果你在几个不同的节点上运行几个 Web 服务器，就必须手动将上转的图片复制到每个 Web 服务器的硬盘上，或者创建某种替代的集中式系统。Mongo 处理这种情况的方法，是用它自己的分布式文件系统，名为 GridFS。这类似我们和 Riak 一起使用的 Luwak。

Mongo 自带一个命令行的工具，与 GridFS 交互。好事情是不需要特别的设置就可以用它。如果利用 mongofiles 命令列出 mongos 管理的分片集群中的文件，会得到一个空列表。

```
$ mongofiles -h localhost:27020 list

connected to: localhost:27020
```

上传一个文件。

```
$ mongofiles -h localhost:27020 put my_file.txt

connected to: localhost:27020
added file: { _id: ObjectId('4d81cc96939936015f974859'), filename: "my_file.txt", \
  chunkSize: 262144, uploadDate: new Date(1300352150507), \
  md5: "844ab0d45e3bded0d48c2e77ed4f3b0e", length: 3067 }
done!
```

如果列出 mongofiles 的内容，我们会看到上传的文件名。

```
$ mongofiles -h localhost:27020 list

connected to: localhost:27020
my_file.txt 3067
```

回到 mongo 控制台，我们可以看到 Mongo 存放数据的集合。

```
> show collections

cities
fs.chunks
fs.files
system.indexes
```

既然它们只是普通的集合，就可以像其他集合一样复制或查询它们。

5.4.6 第 3 天总结

这里结束了对 MongoDB 的研究。今天我们主要关注 Mongo 如何通过副本集，来增强数据的持久性，并通过分片来支持横向伸缩。我们看到了好的服务器配置，以及 Mongo 如何提供 mongos 服务器，作为多节点间自动分片的中继器。

第 3 天作业

求索

1．阅读在线文档中的副本集配置的全部选项。

2．找到创建球面地理（spherical geo）索引的方法。

实践

1．Mongo支持范围形状（即正方形和圆形）。请找出以伦敦为中心，50 英里的正方形之内的所有城市。[1]

2．运行 6 个服务器：3 个服务器构成一个副本集，两个副本集构成两个分片。运行一个配置服务器和 mongos。跨这些服务器运行 GridFS（这是终极考试）。

5.5 总结

我们希望对 MongoDB 的这次尝试已经激发了你的想象力，并向你展示了它如何赢得“其大无比”（humongous）数据库的名号。我们在一章中介绍了许多内容，但像以往一样，我们仅仅触及皮毛。

5.5.1 Mongo的优点

Mongo 的主要优势在于，它能够通过复制和横向伸缩，处理大量的数据（以及大量的请求）。但它还有另一项好处，即非常灵活的数据模型，因为不需要遵从某个模式，可以简单地嵌套任何值，而这在 RDBMS 中通常需要使用 SQL 进行联接。

最后，MongoDB 的设计目标是易于使用。你可能注意到 Mongo 的命令和 SQL 的数据库概念之间的相似性（除了服务器端的联接之外）。这并非偶然，Mongo 受到这么多前对象关系模型（Object Relational Model，ORM）用户青睐，这也是原因之一。它的不同足以

[1] http://www.mongodb.org/display/DOCS/Geospatial+Indexing

挠到许多开发者的痒处，但又没有完全不同，成为令人恐怖的怪物。

5.5.2 Mongo的缺点

Mongo 鼓励反规范化的模式（没有任何模式），这对于一些人可能难以接受。一些开发者发现，关系数据库无情的、严格的约束让人感到放心。将任意类型的任意值插入任意集合，这可能很危险。如果你没想到检查字段名和集合名是否有问题，一个录入错误可能导致几个小时的头痛。如果数据模型已经相当成熟，不会变化，那么 Mongo 的灵活性通常并不重要。

因为 Mongo 关注大型数据集，所以它很适合大型集群，这可能需要花一些努力去设计和管理。在 Riak 中，添加新的节点是透明的、几乎没有痛苦的操作。Mongo 不一样，建立 Mongo 集群需要一些前期思考。

5.5.3 结束语

如果你习惯于 ORM，目前正使用关系数据库来保存数据，Mongo 是很好的选择。我们经常推荐给 Rails、Django 和模型-视图-控制器（Model-View-Controller，MVC）的开发者，因为他们常常在应用层通过模型（model）进行验证和字段管理，也因为不再需要模式迁移（在大多数情况下）。在文档中添加新字段，就像在数据模型中添加新字段一样容易，Mongo 会很高兴地接受新的字段。我们发现，在应用驱动数据集的情况下，对许多常见问题，Mongo 比关系数据库更容易给出自然的解决方案。

第 6 章

CouchDB

轻便的棘轮扳手，可以随身携带用于各种作业。类似于强劲的电钻，你可以像操作螺丝与承孔一样，替换各种尺寸的扳手头。但是，相比电钻需要 120 伏特的交流电，扳手不是心满意足地躺在口袋里，就是在卖力地干活。Apache CouchDB，就是这样的“扳手”，能屈能伸，轻松应对各种规模与复杂度的问题。

CouchDB 是基于 JSON 与 REST 的面向文档的数据库，且堪称该类型数据库的经典。CouchDB 的第一次发布是在 2005 年，该版本的设计考虑了 Web，可随之而来的却是无数的缺陷、错误、故障与瑕疵。所幸的是，CouchDB 最终发展为极其健壮的数据库，为其他大部分数据库所不及。相比其他系统只能容忍偶发的网络中断，CounchDB 甚至能在网络仅偶尔可用时，开足马力正常运作。

与 MongoDB 有点类似，CouchDB 存储由键-值对组成的 JSON 文档，其中，值可取若干类型中的任意一种，包括嵌套任意深度的其他对象。虽然没有自由定义的查询，但是你依然可以查找想要的文档，主要途径是增量式映射规约（incremental mapreduce）生成的索引视图。

6.1　在沙发上放松

CouchDB 无愧于它的口号：放松。相对于只注重大规模集群设施，CouchDB 意在支持各种场景的部署，大到数据中心，小至智能手机。你可以在安卓手机或者 MacBook 上运行 CouchDB，也可以在数据中心中使用。CouchDB 由 Erlang 语言编写，其实现可谓干劲十足——关闭 CouchDB 的唯一方法是结束其进程。而只支持追加的存储模式，实际上使数据不会损坏，易于复制、备份与恢复。

CouchDB 是面向文档的，使用 JSON 作为其存储与通信的语言。正如 Riak，所有对 CouchDB 的调用都运行于 REST 接口之上。复制可以单向也可以双向，可以是自由定义（ad hoc）的也可以是连续的。CouchDB 赋予你足够的灵活性，以决定如何构架、保护与分布你的数据。

CouchDB 与 MongoDB 的比较

本书想回答的一大问题是“CouchDB 与 MongoDB 的差别是什么？”从表面看，CouchDB 与 MongoDB（第 5 章讨论过）相当类似。它们都采用面向文档的数据存储方式，对以 JSON 为数据传输载体的 JavaScript 十分友好。但是，差别也是明显的，从项目理念、实现到可扩展性的特点，都有所不同。在探索 CouchDB 简约之美的过程中，我们会详述这些特性。

在为期三天的“行程”中，我们将探索许多 CouchDB 的亮点与设计选择。一如既往，我们会从单独的 CRUD 命令开始，然后经由映射规约视图，进而讨论索引。与讲解其他数据库一样，我们会导入一些结构化的数据，以此讨论一些进阶概念。最后，我们会用 Node.js 开发一些简单的事件驱动的客户端应用，并学习 CouchDB 的主-主（master-master，相对于主-从 master-slave）复制策略是如何处理更新冲突的。来，开始吧！

6.2 第 1 天：CRUD、Futon[1]与 cURL Redux

今天，CouchDB 的探索之旅将拉开序幕，起点是 CouchDB 中称为 Futon 的 Web 接口，我们会用它执行基本的 CRUD 操作。此后，我们会重拾 cURL——在第 3 章中，用它与 Riak 通信——用于发起 REST 调用。CouchDB 的所有库与驱动程序，其底层实现都归于发送 REST 请求，因此，先理解其 REST 的工作方式，是合乎情理的。

6.2.1 享受 Futon

CouchDB 自带一个名为 Futon 的实用 Web 接口。只要安装并运行 CouchDB，打开 Web 浏览器，访问 `http://localhost:5984/_utils/`，就会显示如图 6-1 所示的 Overview 页面。

[1] Futon 原意为日式床垫，这里为 CouchDB 中名为 Futon 的 Web 接口。——译者注

欢迎来到管理员派对

你可能注意到，在 Futon 页面右栏底部有一条警告消息，告知用户“人人都是管理员”。由于 CouchDB 的目标是成为产品服务器，因此，读到这条消息后，势必要点击“Fix this”链接，新建一个管理员账号，以限制管理员权限。而在此，我们可不做修改，便于后续的各种操作。

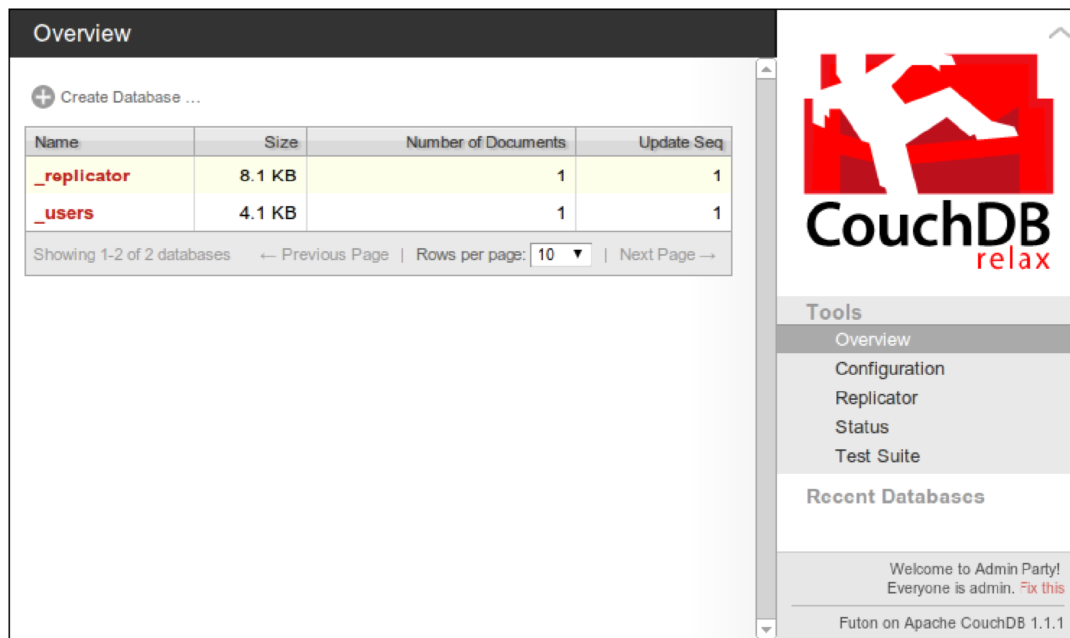


图 6-1 CouchDb Futon:Overview Page

在开始操作文档之前，需要创建一个数据库来容纳它们。我们打算新建一个数据库，存储音乐家的信息，包括其专辑与曲目数据。点击 Create Database 按钮。在弹出框内，输入 music 并点击 Create 按钮。然后，这会自动将你转向 database 的页面。至此，我们就能创建新的或者打开已有的文档了。

在 music 数据库的页面中，点击 New Document 按钮。你会看到一个新的页面，类似图 6-2。

与 MongoDB 一样，每个文档含有一个 JASON 对象，其中又包含称为域（field）的键-值对。CouchDB 中的所有文档都有一个_id 域，它必须是唯一的且不能修改。可以显式地指定_id，即使没有指定，CouchDB 也会自动生成一个。在这里，可以用自动生成的_id，所以点击 Save Document 按钮完成操作即可。

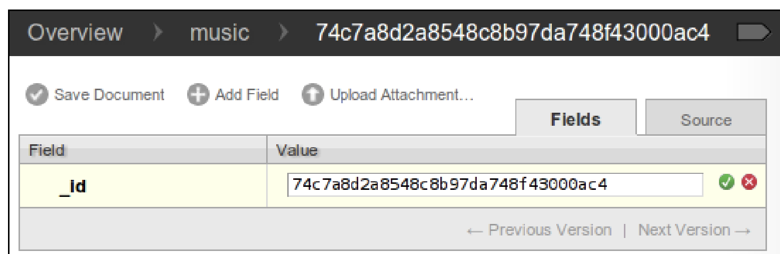


图 6-2　CouchDB Futon：新建一个文档

保存该文档之后，CouchDB 会立刻给它分配另一个名为_rev 的域。这个域会在每次文档变化时，获得新值。这个表示版本的字符串，格式为：整数 + ‘-’ + 唯一的伪随机数字符串。开头的整数就是版本——在此为 1。

以下划线(_)开始的域的名字，对 CouchDB 有特殊含义，像_id 与_rev 就尤其重要。想更新或者删除某个已有的文档，必须提供_id 与相应的_rev。如果发现任何失配，CouchDB 都会拒绝该操作。而这，就是 CouchDB 避免冲突的秘诀——确保只修改最新版本的文档。

CouchDB 没有事务与锁的概念。为了修改已有记录，必须先将它读出，留意其_id 与_rev。然后，提供完整的文档（包括_id 与_rev），以此请求更新。所有操作都是“先到先得”。CouchDB 通过检查并匹配_rev，可以确保你认为你正在修改的版本，不会在你看不到的地方发生变化。

在已经打开的 document 页面中，点击 Add Field 按钮。在 Field 栏中，输入 *name*，在 Value 栏输入 *The Beatles*。点击输入值右侧的绿勾，确保该值没有问题，然后点击 Save Document 按钮。我们会看到_rev 域的值变成以 2 开头了。

CouchDB 不限于存储字符串，它能处理任何嵌套深度的 JSON 结构。再次点击 Add Field 按钮。这次 Filed 设置为 *albums*，Value 域则填入如下内容（这不是完整列表）：

```
[
  "Help!",
  "Sgt. Pepper's Lonely Hearts Club Band",
  "Abbey Road"
]
```

点击 Save Document 按钮后，你会看到图 6-3 所示的页面。

除了名字，还有更多关于专辑的信息，可以继续添加一些。修改 albums 域，用如下的值替换刚输入的那些值：

图 6-3　CouchDB Futon: document with an array value

```
[{
  "title": "Help!",
  "year": 1965
},{
  "title": "Sgt. Pepper's Lonely Hearts Club Band",
  "year": 1967
},{
  "title": "Abbey Road",
  "year": 1969
}]
```

保存文档后，你已经不知不觉扩展了 `albums` 的值——嵌套文档。页面看起来类似图 6-4。

图 6-4　CouchDB Futon：包含深度嵌套值的文档

单击 Delete Document 按钮；正如其名，会将文档从 music 数据库删除它。但是，先别这么做。回到命令行，看看如何通过 REST 与 CouchDB 交互。

6.2.2 用 cURL 执行基于 REST 的 CRUD 操作

与 CouchDB 的所有通信都是基于 REST 的，即通过 HTTP 发送命令。除了 CouchDB，我们还讨论过其他通过 REST 交互的数据库。比如，第 3 章讨论的，Riak 也基于 REST 实现客户端与数据库的所有交互。与 Riak 一样，可以使用命令行工具 cURL，与 CouchDB 通信。

在深入视图（view）话题前，先执行一些基础的 CRUD 操作。作为开始，打开命令行界面，运行下面的命令：

```
$ curl http://localhost:5984/
{"couchdb":"Welcome","version":"1.1.1"}
```

发送 GET 请求（cURL 的默认行为），可以获取 URL 中指定的信息。而访问根目录只能告诉你 CouchDB 已经启动，以及目前正在运行的版本。下面不妨查询之前建立的 `music` 数据库的信息（为便于阅读，命令输出已格式化）：

```
$ curl http://localhost:5984/music/
{
  "db_name":"music",
  "doc_count":1,
  "doc_del_count":0,
  "update_seq":4,
  "purge_seq":0,
  "compact_running":false,
  "disk_size":16473,
  "instance_start_time":"1326845777510067",
  "disk_format_version":5,
  "committed_update_seq":4
}
```

该请求返回的信息包括：数据库中存储了多少文档，服务器运行了多久，以及执行操作的数目。

6.2.3 用 GET 读取文档

要检索某个特定文档，需要将其`_id` 追加到数据库 URL 之后，如下所示：

```
$ curl http://localhost:5984/music/74c7a8d2a8548c8b97da748f43000ac4
{
  "_id":"74c7a8d2a8548c8b97da748f43000ac4",
  "_rev":"4-93a101178ba65f61ed39e60d70c9fd97",
  "name":"The Beatles",
  "albums": [
    {
      "title":"Help!",
      "year":1965
    },{
      "title":"Sgt. Pepper's Lonely Hearts Club Band",
      "year":1967
    },{
      "title":"Abbey Road",
      "year":1969
    }
  ]
}
```

在 CouchDB 中，发送 GET 请求总是安全的。CouchDB 不会因此对文档做任何修改。而要修改数据库，必须使用其他 HTTP 命令，如 `PUT`、`POST` 和 `DELETE`。

6.2.4 用 POST 新建文档

要新建文档，可以使用 POST 方法，但需确保在 HTTP 头 Content-Type 的值为 *application/json*；否则，CouchDB 会拒绝该请求。

```
$ curl -i -X POST "http://localhost:5984/music/" \
  -H "Content-Type: application/json" \
  -d '{ "name": "Wings" }'
HTTP/1.1 201 Created
Server: CouchDB/1.1.1 (Erlang OTP/R14B03)
Location: http://localhost:5984/music/74c7a8d2a8548c8b97da748f43000f1b
Date: Wed, 18 Jan 2012 00:37:51 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 95
Cache-Control: must-revalidate

{
  "ok":true,
  "id":"74c7a8d2a8548c8b97da748f43000f1b",
  "rev":"1-2fe1dd1911153eb9df8460747dfe75a0"
}
```

由 HTTP 响应代码“`201 Created`”，可以知道新建文档请求成功了。而响应体则包含 JSON 对象，该对象中有诸如`_id` 与`_rev` 值的有用信息。

6.2.5　用 PUT 更新文档

PUT 命令用于更新已经存在的文档，也可以创建带有特定_id 的文档。与 GET 一样，PUT 的 URL 包含数据库 URL，且末尾有文档的_id。

```
$ curl -i -X PUT \
  "http://localhost:5984/music/74c7a8d2a8548c8b97da748f43000f1b" \
  -H "Content-Type: application/json" \
  -d '{
    "_id": "74c7a8d2a8548c8b97da748f43000f1b",
    "_rev": "1-2fe1dd1911153eb9df8460747dfe75a0",
    "name": "Wings",
    "albums": ["Wild Life", "Band on the Run", "London Town"]
  }'
HTTP/1.1 201 Created
Server: CouchDB/1.1.1 (Erlang OTP/R14B03)
Location: http://localhost:5984/music/74c7a8d2a8548c8b97da748f43000f1b
Etag: "2-17e4ce41cd33d6a38f04a8452d5a860b"
Date: Wed, 18 Jan 2012 00:43:39 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 95
Cache-Control: must-revalidate

{
  "ok":true,
  "id":"74c7a8d2a8548c8b97da748f43000f1b",
  "rev":"2-17e4ce41cd33d6a38f04a8452d5a860b"
}
```

在 MongoDB 中，可以修改文档的某处；而对于 CouchDB，任何改动，无论多小，都会导致原本的整个文档被重写。我们之前所见的 Futon Web 接口，看似独立修改了单个域，但是当点击 Save 按钮时，其在后台保存了整个文档。

先前提过，_id 与_rev 必须和待更新的文档完全匹配，否则操作就会失败。再次执行相同的 PUT 操作，可以获得感性认识。

```
HTTP/1.1 409 Conflict
Server: CouchDB/1.1.1 (Erlang OTP/R14B03)
Date: Wed, 18 Jan 2012 00:44:12 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 58
Cache-Control: must-revalidate

{"error":"conflict","reason":"Document update conflict."}
```

你会收到一个“HTTP 409 Conflict”应答，以及描述问题的 JSON 对象。这就是 CouchDB 实现一致性的方式。

6.2.6　用 DELETE 移除文档

最后，用 DELETE 操作从数据库中移除文档。

```
$ curl -i -X DELETE \
  "http://localhost:5984/music/74c7a8d2a8548c8b97da748f43000f1b" \
  -H "If-Match: 2-17e4ce41cd33d6a38f04a8452d5a860b"
HTTP/1.1 200 OK
Server: CouchDB/1.1.1 (Erlang OTP/R14B03)
Etag: "3-42aafb7411c092614ce7c9f4ab79dc8b"
Date: Wed, 18 Jan 2012 00:45:36 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 95
Cache-Control: must-revalidate

{
  "ok":true,
  "id":"74c7a8d2a8548c8b97da748f43000f1b",
  "rev":"3-42aafb7411c092614ce7c9f4ab79dc8b"
}
```

尽管被删除的文档即将消失，但 DELETE 操作还是会产生一个新的版本号。值得一提的是，被删除的文档并非真的从磁盘上移除，取而代之的是一个新的空白文档，将该文档标记为已删除。就像更新一样，CouchDB 不就地修改文档。但是，文档还是相当于删除了。

6.2.7　第 1 天总结

我们已经学了如何用 Futon 与 cURL 执行基本的 CRUD 操作，是时候讨论一些更高级的话题了。在第 2 天，我们会深入了解如何创建索引视图。它提供了除了指定_id 值之外另一种检索文档的方式。

第 1 天作业

求索

1. 获取在线 CouchDB HTTP Document API 资料。

2. 我们已经用了 GET、POST、PUT 以及 DELETE。还有其他支持的 HTTP 命令吗？

实践

1. 用 cURL 执行 PUT 命令，新建 music 数据库的文档，其_id 由你指定。
2. 用 cURL 新建数据库，其名字由你指定，然后再用 cURL 将该数据库删除。
3. 用 cURL 新建文档，包含一个文本文档作为附件。最后，编写并执行一个 cURL 请求，仅返回该新建文档的附件。

6.3　第 2 天：创建/查询视图

在 CouchDB 中，通过视图对数据库中的文档一探究竟。总体而言，视图是访问文档的主要方式，但是也不排除某些相对琐碎的场景——比如，我们在第 1 天看到的那些单独的 CRUD 操作。今天，我们会探索如何编写一个创建视图的函数。同样，我们也会学习如何通过 cURL 执行视图的自由定义的查询。最后，我们将导入 music 数据，以使视图更为"夺目"，并演示如何使用 couchrest（一种与 CouchDB 协同工作的 Ruby 库）。

6.3.1　通过视图访问文档

视图包含映射与规约函数，它们用于生成有序的键-值对列表。键或者值都可以是任何合法的 JSON。最为简单的视图称为_all_docs，它会倾整个数据库而出，每个文档都会有对应的条目，且条目以字符串形式的_id 为键值。

要检索数据库中的全部内容，可通过发送 GET 请求获取_all_docs 视图。

```
$ curl http://localhost:5984/music/_all_docs
{
  "total_rows":1,
  "offset":0,
  "rows":[{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"74c7a8d2a8548c8b97da748f43000ac4",
    "value":{
      "rev":"4-93a101178ba65f61ed39e60d70c9fd97"
    }
  }]
}
```

你可以在如上所示的输出中，看到目前为止所创建的文档。得到的响应其实是一个 JSON 对象，包含以若干行为元素的数组（即 rows 数组）。而每行正是包含三个域

的对象。

- id 是文档的_id。
- key 是映射规约函数生成的 JSON 键。
- value 是对应的 JSON 值，也是通过映射规约生成的。

在_all_docs 中，id 和 key 的域是一致的，但是对于自定义的视图，两者几乎从不相同。

视图的默认行为是，不会在返回值中包含每个文档的全部内容。若想检索文档的全部域，就得增加 URL 参数 include_docs=true。

```
$ curl http://localhost:5984/music/_all_docs?include_docs=true
{
  "total_rows":1,
  "offset":0,
  "rows":[{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"74c7a8d2a8548c8b97da748f43000ac4",
    "value":{
      "rev":"4-93a101178ba65f61ed39e60d70c9fd97"
    },
    "doc":{
      "_id":"74c7a8d2a8548c8b97da748f43000ac4",
      "_rev":"4-93a101178ba65f61ed39e60d70c9fd97",
      "name":"The Beatles",
      "albums":[{
        "title":"Help!",
        "year":1965
      },{
        "title":"Sgt. Pepper's Lonely Hearts Club Band",
        "year":1967
      },{
        "title":"Abbey Road",
        "year":1969
      }]
    }
  }]
}
```

于是，你能在输出的对象中看到属性 name 与 albums。对视图的基本结构有了概念，我们着手建立自己的视图。

6.3.2 编写你的第一个视图

既然我们对视图有了粗略的了解，不妨尝试创建我们自己的视图。作为开始，我们会复制_all_docs view 的行为，然后，我们会更进一步，创建复杂的视图，从我们的文档提取更深层的信息并做索引。

如何生成临时视图呢？就像我们在第 1 天所做的操作，打开浏览器，指向 Futon[1]。然后，点击链接，打开 music 数据库。在 music 数据库的页面中，从右上角的 View 下拉菜单中，选择“Temporary view...”。你应该就能看到类似图 6-5 的页面。

左侧的 Map Function 输入框中的代码看起来如下：

```
function(doc) {
  emit(null, doc);
}
```

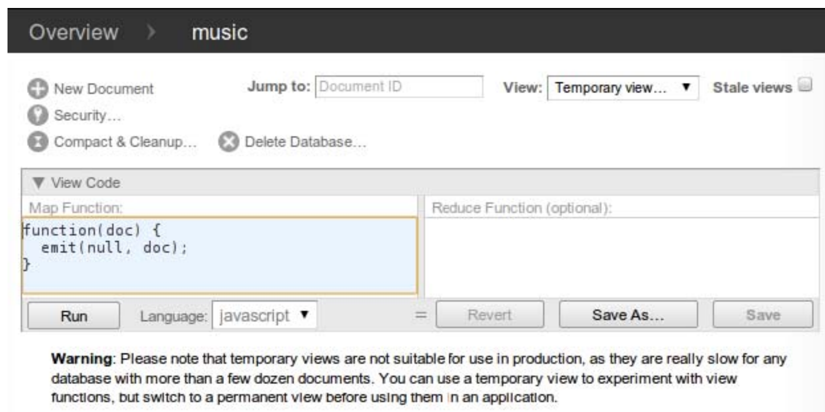


图 6-5 CouchDB Futon：临时视图

若点击 Map Function 下的 Run 按钮，CouchDB 会为数据库中的每个文档执行一次这个函数，每次都会将当前文档作为参数 doc 传入。这个过程会生成一个单行的表，如下所示：

键	值
null	{_id: "74c7a8d2a8548c8b97da748f43000ac4", _rev: "4-93a101178ba65f61ed39e60d70c9fd97", name: "The Beatles", albums: [{title: "Help!", year: 1965}, {title: "Sgt. Pepper's Lonely Hearts Club Band", year: 1967}, {title: "Abbey Road", year: 1969}]}

[1] http://localhost:5984/_utils/

这个输出或者所有视图的秘密就在于名为 emit()的函数（它与 MongoDB 中的同名函数起相同的作用）。emit 函数有两个参数：key 和 value。一个给定的映射（map）函数可以对某个特定的文档调用 emit 函数一次、多次或者根本不调用。在前面的实例中，映射函数就发布了键-值对 null/doc。正如我们在输出表里看到的，键的确是 null，而值则与第 1 天直接用 cURL 请求所获得的对象是同一个。

要编写一个与_all_docs 作用相同的映射器，需要适当修改 emit 函数。记得吗？_all_docs 发布文档的_id 域作为 key，仅包含_rev 域的简单对象作为 value。明确了这一点，修改 Map 函数的代码为以下代码，然后单击 Run 按钮。

```
function(doc) {
  emit(doc._id, { rev: doc._rev });
}
```

现在，输出应该类似于下面这张表，与我们先前通过_all_docs 枚举记录时，所看到的键-值对一样：

键	值
null	{rev: "4-93a101178ba65f61ed39e60d70c9fd97"} "74c7a8d2a8548c8b97da748f43000ac4"

注意，未必要用 Futon 生成临时视图。可以向_temp_view 处理程序发送 POST 请求。在这个实例中，把修改过的 map 函数作为 JSON 对象，放到请求体中即可。

```
$ curl -X POST \
  http://localhost:5984/music/_temp_view \
  -H "Content-Type: application/json" \
  -d '{"map":"function(doc){emit(doc._id,{rev:doc._rev});}"}'
{
  "total_rows":1,
  "offset":0,
  "rows":[{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"74c7a8d2a8548c8b97da748f43000ac4",
    "value":{
      "rev":"4-93a101178ba65f61ed39e60d70c9fd97"
    }
  }]
}
```

于是，取得的响应与从_all_docs 获得的完全一样。但是，如果增加参数 include_docs=true，又会怎样呢？我们一起试试看。

```
$ curl -X POST \
```

```
  http://localhost:5984/music/_temp_view?include_docs=true \
  -H "Content-Type: application/json" \
  -d '{"map":"function(doc){emit(doc._id,{rev:doc._rev});}"}'
{
  "total_rows":1,
  "offset":0,
  "rows":[{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"74c7a8d2a8548c8b97da748f43000ac4",
    "value":{
      "rev":"4-93a101178ba65f61ed39e60d70c9fd97"
    },
    "doc":{
      "_id":"74c7a8d2a8548c8b97da748f43000ac4",
      "_rev":"4-93a101178ba65f61ed39e60d70c9fd97",
      "name":"The Beatles",
      "albums":[...]
    }
  }]
}
```

这次，其他的域并没有添加到 `value` 对象中，而把一个名为 `doc` 的单独属性加入到 row 中，包含了完整的文档。

定制视图可以发布任何数据，甚至 null。提供一个单独的 `doc` 属性，可以防止 row 的值与文档相混淆。下一步将讨论如何保存视图，以便 CouchDB 对结果进行索引。

6.3.3　将视图另存为“设计文档”

当 CouchDB 生成临时视图时，它必须为数据库中的每个文档，执行既有的映射函数。这极其耗费资源，占用大量的计算力，并且十分耗时。所以，只有出于开发目的，才应该使用临时视图。对于实际的产品环境，应当把视图存储在“设计文档”（design document）中。

“设计文档”是数据库中实际存在的文档，正如早前创建的 Beatles 文档。于是，它可以显示在视图中，也能以常用方式复制到其他 CouchDB 服务器中。在 Futon 中，将临时视图另存为“设计文档”，只需单击 Save As...按钮，然后填写 Design Document 与 View Name 输入框。

“设计文档”的 ID 总是以`_design/`开头，并且包含一个或者多个视图。视图的名字则必须与同一个“设计文档”中的其他视图不同。而决定哪个视图属于哪个“设计文档”，主要取决于特定的应用程序以及看问题的角度。一般来说，你该关注视图对你的数据做了

什么操作，以此判断是否将视图分组。随着创建更多有意思的视图，我们会看到一些相关的例子。

6.3.4　由 Name 查找 Artists

既然有了创建视图的基础，我们不妨开发一个适用于某个应用的视图。记得 `music` 数据库吗？它存储音乐家的信息，包含一个描述乐队名的 `name` 域。使用普通的 GET 方法或者`_all_docs view` 视图，可以通过其`_id` 值访问文档，但是更让人感兴趣的是通过 name 域查询乐队。

换句话说，今天我们已经可以通过条件`_id` 等于 `74c7a8d2a8548c8b97da748f43000ac4` 来查找文档，但是如何由 `name` 等于 The Beatles 进行查找呢，为此，需要一个视图。在 Futon 里，回到 Temporary View 页面，输入以下 Map 函数代码，然后点击 Run 按钮。

```
couchdb/artists_by_name_mapper.js
function(doc) {
  if ('name' in doc) {
    emit(doc.name, doc._id);
  }
}
```

这个函数检查当前文档是否有 `name` 域，如果有，就发布其 name 域与这个文档的`_id` 作为键-值对。这将生成一张表，如下所示：

键	值
"The Beatles"	"74c7a8d2a8548c8b97da748f43000ac4"

单击 Save As...按钮；再单击 Design Document，输入 artists，然后输入 *by_name* 作为 View Name。单击 Save 按钮保存修改。

6.3.5　由 name 查找 albums

通过 name 查找 artist 相当有用，但是这还不够。现在，我们要建立一个视图，可以用于查找 album。这会是映射函数为每个文档发布多个结果的第一次尝试。

再次返回 Temporary View 页面；然后输入以下映射器。

```
couchdb/albums_by_name_mapper.js
```

```
function(doc) {
  if ('name' in doc && 'albums' in doc) {
    doc.albums.forEach(function(album){
      var
        key = album.title || album.name,
        value = { by: doc.name, album: album };
      emit(key, value);
    });
  }
}
```

该函数检查当前文档是否含有 `name` 与 `albums` 域。如果有，则为每张 album 发布一个键-值对，其中键为 album 的 title 或者 name，值为一个复合对象——包含 artist 的 name 与原始的 album 对象。这会生成如下所示的表：

键	值
"Abbey Road"	{by: "The Beatles", album: {title: "Abbey Road", year: 1969}}
"Help!"	{by: "The Beatles", album: {title: "Help!", year: 1965}}
"Sgt. Pepper's Lonely Hearts ClubBand"	{by: "The Beatles", album: {title: "Sgt. Pepper's Lonely HeartsClub Band", year: 1967}}

正如我们对 Artist By Name 视图做的操作，点击 Save As...按钮。这次对于 Design Document 输入 *albums*，而对于 View Name 输入 *by_name*。点击 Save 按钮保存修改。现在来看看如何查询这些文档。

6.3.6　查询自定义的 Artist 与 Album 视图

既然我们已经保存了两个自定义的“设计文档”，现在就该回到命令行，用 `curl` 命令查询它们。不妨从 Artists By Name 视图开始，执行如下命令：

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name
{
  "total_rows":1,
  "offset":0,
  "rows":[{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"The Beatles",
    "value":"74c7a8d2a8548c8b97da748f43000ac4"
  }]
}
```

查询视图，需要构造这样的路径`/<database_name>/_design/<design_ doc>/_view/<view_ name>`，替代相应的部分即可。在该实例中，在 `music` 数据库的 `artists`“设计文档”中，查询 `by_name` 视图。意料之中，输出包含了一个文档，以乐队名为键。

下一步，试着查找 Albums By Name 视图：

```
$ curl http://localhost:5984/music/_design/albums/_view/by_name
{
  "total_rows":3,
  "offset":0,
  "rows":[{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"Abbey Road",
    "value":{
      "by":"The Beatles",
      "album":{
        "title":"Abbey Road",
        "year":1969
      }
    }
  },{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"Help!",
    "value":{
      "by":"The Beatles",
      "album":{
        "title":"Help!",
        "year":1965
      }
    }
  },{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"Sgt. Pepper's Lonely Hearts Club Band",
    "value":{
      "by":"The Beatles",
      "album":{
        "title":"Sgt. Pepper's Lonely Hearts Club Band",
        "year":1967
      }
    }
  }]
}
```

CouchDB 会保证记录以所发布的键的字母顺序呈现。实质上，这就是 CouchDB 提供的索引。在设计视图时，选择发布什么键尤为重要，因为这会决定进行排序的依据。

以这种方式请求视图会返回整个数据聚合，但是，如果我们只想要一个子集呢？一种方法是使用 URL 参数 key。指定 key 后，只有完全匹配的行才会返回。

```
$ curl 'http://localhost:5984/music/_design/albums/_view/by_name?key="Help!"'
{
  "total_rows":3,
  "offset":1,
  "rows":[{
    "id":"74c7a8d2a8548c8b97da748f43000ac4",
    "key":"Help!",
    "value":{
      "by":"The Beatles",
      "album":{"title":"Help!","year":1965}
    }
  }]
}
```

注意响应中的 total_rows 与 offset 域。total_rows 域表示视图中记录的总数，不只是这个请求所返回的子集。offset 域则告诉我们，在全部记录中，第一条记录出现位置。基于这两个数字，以及 rows 的长度，我们能够计算在视图中在该位置前后还有多少记录。

请求视图可以分为基于 key 参数的若干种方式，但要在实际操作中演示，我们需要更多数据。

6.3.7 使用 Ruby 将数据导入 CouchDB

不管使用什么数据库，导入数据总是无法回避的问题。CouchDB 也不例外。本节会用 Ruby 将结构化数据导入 music 数据库。通过这个实例，你会看到如何在 CouchDB 中实施大规模的数据导入，同时，我们也将得到一个很棒的数据池，供我们建立进阶视图时使用。

我们使用取自 Jamendo.com（一个专门提供免费授权音乐的网站[1]）的音乐数据。Jamendo 以结构化 XML 的形式提供所有的艺术家、专辑以及曲目的数据，这对于 CouchDB 这样的面向文档的数据库来说，再理想不过了。

翻到 Jamendo 的 NewDatabaseDumps 页面[2]，下载文件 dbdump_artistalbumtrack.xml.gz[3]，这个压缩文件只有 15MB。解析 Jamendo 的 XML 文件，需要用到 libxml-ruby gem。

---

1 http://www.jamendo.com/
2 http://developer .jamendo.com/en/wiki/NewDatabaseDumps
3 http://img.jamendo.com/data/dbdump_artistalbumtrack.xml.gz

不必编写我们自己的 Ruby-CouchDB 驱动程序，也不必直接发送 HTTP 请求，我们使用的是一个颇为流行的名为 `couchrest` 的 Ruby gem，它将这些调用转化成方便使用的 Ruby API。我们只会使用 API 中的某些方法，但是如果你想继续在项目中使用这个驱动程序，相关文档十分详细[1]。

在命令行中，安装必要的 gem：

```
$ gem install libxml-ruby couchrest
```

就像我们在第 4 章中对数据所做的一样，我们使用一个 SAX 风格的解析器来处理文档，当数据从标准输入进来的时候，按顺序插入。代码如下：

```
couchdb/import_from_jamendo.rb
① require 'rubygems'
  require 'libxml'
  require 'couchrest'

  include LibXML

② class JamendoCallbacks

    include XML::SaxParser::Callbacks

③ def initialize()

      @db = CouchRest.database!("http://localhost:5984/music")

      @count = 0
      @max = 100 # maximum number to insert
      @stack = []

      @artist = nil
      @album = nil
      @track = nil
      @tag = nil

      @buffer = nil

    end

④   def on_start_element(element,attributes)
      case element
      when 'artist'
        @artist = { :albums => [] }
```

1 http://rdoc.info/github/couchrest/couchrest/master/

```
        @stack.push @artist
      when 'album'
        @album = { :tracks => [] }
        @artist[:albums].push @album
        @stack.push @album
      when 'track'
        @track = { :tags => [] }
        @album[:tracks].push @track
        @stack.push @track
      when 'tag'
        @tag = {}
        @track[:tags].push @tag
        @stack.push @tag
      when 'Artists', 'Albums', 'Tracks', 'Tags'
        # ignore
      else
        @buffer = []
      end
    end

⑤  def on_characters(chars)
      @buffer << chars unless @buffer.nil?
    end

⑥  def on_end_element(element)
      case element
      when 'artist'
        @stack.pop

        @artist['_id'] = @artist['id'] # reuse Jamendo's artist id for doc _id
        @artist[:random] = rand

        @db.save_doc(@artist, false, true)
        @count += 1
        if !@max.nil? && @count >= @max
          on_end_document
        end
        if @count % 500 == 0
          puts " #{@count} records inserted"
        end

      when 'album', 'track', 'tag'
        top = @stack.pop
        top[:random] = rand
      when 'Artists', 'Albums', 'Tracks', 'Tags'
        # ignore
      else
        if @stack[-1] && @buffer
          @stack[-1][element] = @buffer.join.force_encoding('utf-8')
          @buffer = nil
        end
```

```
      end
    end

      def on_end_document()
      puts "TOTAL: #{@count} records inserted"
      exit(1)
    end

  end

⑦ parser = XML::SaxParser.io(ARGF)
  parser.callbacks = JamendoCallbacks.new
  parser.parse
```

① 作为开始，引入 rubygems 模块以及需要的某些 gems。

② 使用 LibXML 的标准方式是定义回调（callback）类。这里定义一个 JamendoCallbacks 类，为不同的事件（events）封装 SAX 处理程序。

③ 初始化阶段，要做的第一件事情是用 CouchRest API 连接本地的 CouchDB 服务器，然后创建 music 数据库（如果原本不存在 music 数据库）。之后，设置一些实例变量，以便在解析时存储状态信息。注意，如果把参数@max 设置为 nil，那么所有文档都会导入，而不是前 100 个文档。

④ 一旦开始解析，on_start_element()方法会处理任何开始标签。这里，我们会看到某些非常有趣的标签，比如<artist>、<album>、<track>以及<tag>。特别地，我们会忽略某些容器元素——<Artists>、<Albums>、<T racks>与<T ags>——而对于其他元素，我们会将它们作为在最近的容器项上设置的属性。

⑤ 解析器无论何时碰到字符数据，都会将它们存到缓冲区中，以便作为属性加到当前的容器元素中（@stack 的末尾）。

⑥ 在 on_end_element()方法中，有许多有趣的东西。这里，通过将当前容器元素弹出栈，来将它关闭。如果标签变量关闭了<artist>元素，就能用方法@db.save_doc()将文档存入 CouchDB 了。对任何容器元素，也会增加一个随机属性，其中包含新生成的随机数。我们会在稍后随机选择曲、专辑和艺术家时用到它。

⑦ Ruby 的 ARGF 流会结合标准输入与任何在命令行中指定的文件。我们将 ARGF 流定向到 LibXML，并指定 JamendoCallbacks 类的实例，处理这些标记——开始标记、结束标记以及字符数据。

把解压后的 XML 管道连接到这个导入脚本，就可以运行脚本了。

```
$ zcat dbdump_artistalbumtrack.xml.gz | ruby import_from_jamendo.rb
```

```
TOTAL: 100 records inserted
```

当输入完毕，回到命令行，我们可以看到新的视图。首先，查询一些艺术家。URL 参数 `limit` 规定了我们希望返回的文档数目。

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?limit=5
{"total_rows":100,"offset":0,"rows":[
{"id":"370255","key":"\"\"ATTIC\"\"","value":"370255"},
{"id":"353262","key":"10centSunday","value":"353262"},
{"id":"367150","key":"abdielyromero","value":"367150"},
{"id":"276","key":"AdHoc","value":"276"},
{"id":"364713","key":"Adversus","value":"364713"}
]}
```

上个请求查询了 artists 列表中很靠前的文档。要跳到列表中间的位置，可以使用 `startkey` 参数：

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?\
limit=5\&startkey=%22C%22
{"total_rows":100,"offset":16,"rows":[
{"id":"340296","key":"CalexB","value":"340296"},
{"id":"353888","key":"carsten may","value":"353888"},
{"id":"272","key":"Chroma","value":"272"},
{"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"},
{"id":"364714","key":"Daringer","value":"364714"}
]}
```

上面这条命令，从首字母为 C 的艺术家开始。而指定 `endkey` 参数，则提供了另一种限制返回内容的方式。这里限定只查询首字母在 C～D 之间的艺术家。

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?\
startkey=%22C%22\&endkey=%22D%22
{"total_rows":100,"offset":16,"rows":[
{"id":"340296","key":"CalexB","value":"340296"},
{"id":"353888","key":"carsten may","value":"353888"},
{"id":"272","key":"Chroma","value":"272"},
{"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"}
]}
```

要获取反序的记录，可以使用 URL 参数 `descending`。当然，记得交换参数 `startkey` 与 `endkey`。

```
$ curl http://localhost:5984/music/_design/artists/_view/by_name?\
startkey=%22D%22\&endkey=%22C%22\&descending=true
```

```
{"total_rows":100,"offset":16,"rows":[
{"id":"351138","key":"Compartir D\u00f3na Gustet","value":"351138"},
{"id":"272","key":"Chroma","value":"272"},
{"id":"353888","key":"carsten may","value":"353888"},
{"id":"340296","key":"CalexB","value":"340296"}
]}
```

还有很多其他的 URL 参数，可用于修改视图请求；这里介绍的是最常见也最常用的一些。URL 参数中，有些必须与分组（grouping）一起使用，而这出自 CouchDB 映射规约视图的规约部分。我们将在明天探索这部分内容。

6.3.8　第 2 天总结

今天介绍了一些有意思的话题。我们学习了如何在 CouchDB 中创建基本视图，并保存到“设计文档”，还探索了查询视图的多种方式，以获得某些经过索引的数据。而通过使用 Ruby 与名为 `couchrest` 的流行 gem，我们成功导入了结构化数据，并用这些数据支撑了我们之后创建的视图。展望明天，我们会创建更高级的视图，扩展已有的这些知识点，而方式则是引入规约器（reducer）并使用其他 CouchDB 支持的 API。

求索

1．我们知道 emit()方法能够输出字符串类型的键。其他数据类型，是否支持呢？你若发布一个作为键的数组，又会如何？

2．寻找一组可用于请求视图的 URL 参数（就像 limit 与 startkey），并了解其用途。

实践

1．导入脚本 import_from_jamendo.rb 会为每个 artist 增加一个 random 属性，以分配随机数。创建映射函数会发布键-值对，其中键就是随机数，值则是 band 的名字。根据以上描述，新建视图 artist 并保存到名为_design/random 的“设计文档”。

6.4　第 3 天：进阶视图、Changes API 以及复制数据

在第 1～2 天，我们学习了如何执行基本的 CRUD 操作，以及与视图交互查找数据。基于这些经验，今天我们会更深入了解视图，解析映射规约架构的规约部分。之后，我们会用 JavaScript 编写一些 Node.js 应用，以运用 CouchDB 独特的 Changes API。最后，我们将讨论复制数据，以及 CouchDB 是如何处理数据冲突的。

6.4.1 用规约器创建进阶视图

基于映射规约的视图为我们提供了方法，让我们能够驾驭索引以及聚合工具（aggregation facility）。在第 2 天，所有的视图都只包含映射器（mapper）。现在，加入规约器（reducer），对前一天导入的 Jamendo 数据，开发新的功能。

Jamendo 数据的一大优点是其可供挖掘的深度。艺术家有专辑，专辑有曲目。曲目，又有包括标签在内的属性。我们会将注意力转移到标签，看看能否编写一个收集标签并计数的深入化的视图。

首先，回到 Temporary View 页面，然后输入如下的 map 函数：

```
couchdb/tags_by_name_mapper.js
function(doc) {
  (doc.albums || []).forEach(function(album){
    (album.tracks || []).forEach(function(track){
      (track.tags || []).forEach(function(tag){
        emit(tag.idstr, 1);
      });
    });
  });
}
```

该函数挖掘了 artist 文档，深入到每张专辑，每首曲目，最后触及每个标签。而对每个标签，它发布了键-值对；其中，键为标签的 `idstr` 属性（标签的字符串形式的表示，如“rock”），值为数值 1。

有了 map 函数，在 Reduce Function 中输入如下代码：

```
couchdb/simple_count_reducer.js
function(key, values, rereduce) {
  return sum(values);
}
```

这段代码的作用仅仅是计算值列表中数字的总和——运行视图后，马上会讨论这个话题。最后，点击 Run 按钮。输出应该如下所示：

键	值
"17sonsrecords"	1
"17sonsrecords"	1
"17sonsrecords"	1
"17sonsrecords"	1

续表

键	值
"17sonsrecords"	1
"acid"	1
"acousticguitar"	1
"acousticguitar"	1
"action"	1
"action"	1

运行结果在意料之中。正如我们在 mapper 函数中所见，值总是 1；而键则重复出现若干次，其次数与该标签用于曲目的数目一致。

注意了，输出表右上角有个 Reduce 复选框。选中它，再看看输出表。现在，它应该看起来如下所示：

键	值
"17sonsrecords"	5
"acid"	1
"acousticguitar"	2
"action"	2
"adventure"	3
"aksband"	1
"alternativ"	1
"alternativ"	3
"ambient"	28
"autodidacta"	17

这是怎么回事？简而言之，规约器通过合并将输出规约了，比如，依据 Reducer Function 合并映射行。从概念上来看，CouchDB 映射规约引擎与我们之前所见的其他映射规约器无异（见 3.3.2 节和 5.3.4 节（以及 Finalize））。这里，我们特别从较高层次概括了 CouchDB 建立一个视图所采取步骤如下：

（1）将文档发送给映射函数；

（2）收集所有发布的值；

（3）根据键，将发布的行排序；

（4）将键相同的各行发送给规约函数；

（5）如果试图在一次调用中完成全部规约，数据可能会多到无法处理；这时，可以再次调用规约函数，当然，该次调用处理的是上次规约的结果；

（6）重复递归调用规约函数，直到没有重复的键。

CouchDB 中的规约函数有三个参数：`key`、`values` 以及 `rereduce`。第一个参数，（key），是以元组（tuple）为元素的数组——元组包含：映射器发布的键，以及生成这个键的文档的`_id`。第二个参数（`values`），是由值所组成的数组，这些值对应于第一个参数中的键。

第三个参数，（`rereduce`），是一个布尔值。如果该次调用是规约的规约，则该值为 true。也就是说，除了接收映射器发布的键与值，还接收上次规约器调用生成的结果。在这种情况下，参数 `key` 会是 `null`。

6.4.2　规约器调用详解

让我们看一个实例，该例子基于我们刚才看到的输出。考虑文档（artists），其中包含了标记为“ambient”的 tracks。映射器作用于这些文档，并发布键-值对，形如“ambient”/1。

某些时候，映射器发布了足够多的键值对，于是 CouchDB 调用规约器，如下所示：

```
reduce(
  [["ambient",id1],["ambient",id2],...],        //key 都相同
  [1,1,...],                                   //value 都是 1
  false                                         //rereduce 是 false,即非规约的规约
)
```

记得吗？在规约函数中，计算了值的总和——`sum()`。因为所有值都是 1，所以这里的总和，实际是标记为“ambient”的曲目的总数。CouchDB 会保存该返回值，供进一步处理。对于这例子，把这个总数计为 10。

稍后，等待 CouchDB 执行这些调用若干次，就会通过一个规约的规约，合并中间规约器结果：

```
reduce(
  null,                    //key 数组为 null
  [10,10,8],          //value 为之前规约器调用的输出
  true                     //rereduce 为 true
)
```

规约函数再次对 `values` 调用 `sum()`。这次，`values` 的和为 28。规约器调用可以是递归的。只要有规约没有完成，就会一直继续下去，直到所有的中间结果都合并成一个值。

大多数映射规约系统，包括其他数据库（如本书讨论过的 Riak 与 MongoDB）所使用的，都会在工作结束后丢弃映射规约的输出。在那些系统中，把映射规约视为获取结果的手段——每当有需求时，就要执行，每次都要从头开始。而 CouchDB 并非如此。

一旦视图编码为“设计文档”，CouchDB 就会保存映射器与规约器的中间值，直到文档发生变化，中间数据不再有效。这时，CouchDB 会增量地运行映射器与规约器，为更新的数据更正映射规约的中间结果。于是，CouchDB 不会每次从头开始，重复每项计算。这是 CouchDB 的“天赋”。CouchDB 不会丢弃中间数据值，因此，能够以映射规约作为主要的索引机制。

6.4.3 监控 CouchDB 的变化

CouchDB 的增量式映射规约无疑是一个创新的特性；这是 CouchDB 区别于其他数据库的特性之一。我们将探索的下一特性是 Changes API。这个接口提供了监控数据库变化并立刻更新的机制。

Changes API 使 CouchDB 成为记录系统的完美候选。设想有一个多数据库系统，数据从各处流入，而其他系统需要时时保持更新（事实上，我们会在 8.4 节讨论相关内容）。相关例子会包含搜索引擎，由如下技术或者项目支持：Lucene、ElasticSeach、实现于 memcached 之上的缓存层，或者 Redis。同样，你也可能要运行不同的维护脚本，以应对数据变化——执行诸如数据库压缩或者远程备份的任务。简而言之，这个简单的 API 带来了无穷的可能性。今天，我们就来学习如何利用它。

为使用这个 API，我们将用 Node.js 开发一些简单的客户端应用。Node.js[1]是基于 V8 JavaScript 引擎的服务器端 JavaScript 平台——Google 的 Chrome 浏览器使用的就是 V8。由于 Node.js 是事件驱动的，且与之协同工作的代码是 JavaScript 编写的，因此，十分适合与 CouchDB 整合。如果你还没有 Node.js，浏览 Node.js 的网站，安装其稳定版（我们使用的是 0.6 版）。

Changes API 有三种：轮询（polling）、长轮询（long-polling）以及连续（continuous）。我们会依次讨论每一项。与往常一样，我们会先使用 cURL 小试身手，然后再采用编程方式。

---

[1] http://nodejs.org/

1. 用于变更的 cURL

访问 Changes API 的最简单方法是利用轮询接口。打开命令行，尝试如下命令（为简单起见，这里展示的是输出片段；所以可能与你得到的输出所有不同）：

```
$ curl http://localhost:5984/music/_changes
{
  "results":[{
    "seq":1,
    "id":"370255",
    "changes":[{"rev":"1-a7b7cc38d4130f0a5f3eae5d2c963d85"}]
  },{
    "seq":2,
    "id":"370254",
    "changes":[{"rev":"1-2c7e0deec3ffca959ba0169b0e8bfcef"}]
  },{
    ... 97 more records ...
  },{
    "seq":100,
    "id":"357995",
    "changes":[{"rev":"1-aa649aa53f2858cb609684320c235aee"}]
  }],
  "last_seq":100
}
```

当向_changes 发送 GET 请求，而不带任何参数时，CouchDB 会返回它所能提供的全部。正如访问视图，可以指定参数 limit，以请求数据的子集，而添加参数 include_docs=true 会使响应包含完整的文档。

通常，你并不想获取从最初算起的所有变化。你往往更希望得到自上次检查后所发生的变化。为此，需要使用 since 参数。

```
$ curl http://localhost:5984/music/_changes?since=99
{
  "results":[{
    "seq":100,
    "id":"357995",
    "changes":[{"rev":"1-aa649aa53f2858cb609684320c235aee"}]
  }],
  "last_seq":100
}
```

如果指定的 since 参数值大于最后的序号，就会得到一个空的响应：

```
$ curl http://localhost:5984/music/_changes?since=9000
  {
```

```
  "results":[
  ],
  "last_seq":9000
}
```

使用这个方法，客户端应用会定期检查，以发现是否发生新的变化，然后根据不同的应用，采取不同的行为。

如果你跟踪数据的更新，又能够容忍一定的延迟，那么轮询是不错的方案。如果数据更新相对较少，尤其适合轮询。比如，如果你要获取博客文章，每五分钟轮询一次就可以了。

如果你想快些更新，又没有重新打开连接的开销，那么长轮询是更好的选择。当指定 URL 参数 `feed=longpoll` 时，CouchDB 会保持连接一段时间，等待更新，然后完成响应。试试下面的命令：

```
$ curl 'http://localhost:5984/music/_changes?feed=longpoll&since=9000'
{"results":[
```

你应该只能看到 JSON 响应的开始部分。如果终端打开足够长时间，CouchDB 最终会关闭连接，以此作为结束：

```
],
"last_seq":9000}
```

从开发角度看，编写驱动程序，使用轮询监控 CouchDB 变更，等效于长轮询。本质的区别只是 CouchDB 会保留连接处于打开状态多久。现在，不妨将注意力放到编写 Node.js 应用上，以监控并利用数据变化订阅内容。

2. 用 Node.js 轮询变化

因为 Node.js 是强大的事件驱动系统，所以 CouchDB 监控亦会坚持这个原则。无论 CouchDB 何时报告文档变化，驱动程序都会监控订阅与发布的变化事件。首先，我们会概览这个驱动程序的构架，谈论它的主要部件，然后填入订阅的具体细节。

事不宜迟，以下就是监控程序的框架，以及关于其功能的简单讨论：

```
couchdb/watch_changes_skeleton.js
var
  http = require('http'),
  events = require('events');

/**
 * create a CouchDB watcher based on connection criteria;
```

```
 * follows node.js EventEmitter pattern, emits 'change' events.
 */
① exports.createWatcher = function(options) {
②  var watcher = new events.EventEmitter();

    watcher.host = options.host || 'localhost';
    watcher.port = options.port || 5984;
    watcher.last_seq = options.last_seq || 0
    watcher.db = options.db || '_users';
③   watcher.start = function(){
      // ...特定于订阅内容的实现
    };

    return watcher;

};

//作为主脚本运行,开始监控 CouchDB 的变化

④ if (!module.parent){
    exports.createWatcher({
      db: process.argv[2],
      last_seq: process.argv[3]
    })
      .on('change', console.log)
      .on('error', console.error)
      .start();
}
```

① exports 是由 CommonJS Module API（实现了 Node.js）提供的标准对象。为 exports 增加 createWatcher()方法，使它能够用于其他 Node.js 脚本（这些脚本想把 exports 作为类库使用）。options 参数允许调用者指定希望监控的数据库以及覆盖其他的连接设置。

② createWatcher()方法生成 EventEmitter 对象，调用者可以用这个对象来监听变化事件。EventEmitter 的这些能力包括，调用 on()方法可以监听事件，调用 emit()方法可以触发事件。

③ watcher.start()负责发送 HTTP 请求，以监控 CouchDB 的变化。当文档发生变化时，监控器会发布相应的变化事件。所有特定于订阅内容的实现都在这个方法内。

④ 最后一段代码定义了，如果该脚本直接从命令行调用，会做些什么。在这个例子中，脚本会调用 createWatcher()方法，然后在返回的对象上设置监听者（监听什么，之后做什么），并把结果置于标准输出。在命令行中，可以设置连接哪个数据库，从哪个序号

ID 开始。

到目前为止，这些代码中没有特定于 CouchDB 的内容，都只是 Node.js 的实现方式。你可能对这些代码感到陌生，尤其是你之前从未开发过事件驱动的服务器技术，然而，随着本书的进度，我们会越来越多地使用这项技术。

有了实现架构之后，不妨加入代码，通过长轮询与发布事件的方式，连接到 CouthDB。下面就是 `watcher.start()`方法内的代码。参考前面的代码结构，所完成的新文件名为 `watch_changes_longpolling.js`。

couchdb/watch_changes_longpolling_impl.js

```
var
① http_options = {
    host: watcher.host,
    port: watcher.port,
    path:
      '/' + watcher.db + '/_changes' +
      '?feed=longpoll&include_docs=true&since=' + watcher.last_seq
  };

② http.get(http_options,function(res) {
    var buffer = '';
    res.on('data', function (chunk) {
      buffer += chunk;
    });
    res.on('end', function() {
③     var output = JSON.parse(buffer);
      if (output.results) {
        watcher.last_seq = output.last_seq;
        output.results.forEach(function(change){
          watcher.emit('change', change);
        });
        watcher.start();
      } else {
        watcher.emit('error', output);
      }
    })
  })
  .on('error', function(err) {
    watcher.emit('error', err);
  });
```

① 这个脚本所做的第一件事情就是在请求中设置 `http_options` 配置对象。`path` 则指向我们用过的相同 `_changes` URL，而 `feed` 设置为 `longpoll`，`include_docs=true`。

② 此后，脚本会调用 http.get()——Node.js 的库方法，它会依据设置发送一个 GET 请求。该方法的第二个参数是一个回调方法，它的功能是收到一个 `HTTPResponse`。响应对象会发布 `data` 事件，作为传回的内容，即会加到 `buffer` 中。

③ 最后，当响应对象发布 `end` 事件，会解析缓冲区（其中包含 JSON）。由此，得到新的 `last_seq` 值，发布一个 `change` 事件，然后再次调用 `watcher.start()`，等待下一次数据变化。

在命令行模式运行这个脚本，如下所示（简洁起见，截取了输出的片段）：

```
$ node watch_changes_longpolling.js music
{ seq: 1,
  id: '370255',
  changes: [ { rev: '1-a7b7cc38d4130f0a5f3eae5d2c963d85' } ],
  doc:
   { _id: '370255',
     _rev: '1-a7b7cc38d4130f0a5f3eae5d2c963d85',
     albums: [ [Object] ],
     id: '370255',
     name: '""ATTIC""',
     url: 'http://www.jamendo.com/artist/ATTIC_(3)',
     mbgid: '',
     random: 0.4121620435325435 } }
{ seq: 2,
  id: '370254',
  changes: [ { rev: '1-2c7e0deec3ffca959ba0169b0e8bfcef' } ],
  doc:
   { _id: '370254',
     _rev: '1-2c7e0deec3ffca959ba0169b0e8bfcef',
  ... 98 more entries ...
```

应用运行良好！为每个文档输出一条记录后，进程会继续运行，轮询 CouchDB 之后的变化。

可以随时在 Futon 里直接修改文档，或者在 `import_from_jamendo.rb` 上增加 `@max` 值，并再次运行。你会看到在命令行中看到相应的变化。接下来会讨论如何开足马力，使用连续的事件通知，以实现更快的更新。

6.4.4　连续监控变化

`_changes` 服务提供的轮询与长轮询订阅都会产生正确的 JSON 作为结果。而连续

的订阅有所不同——单独发送每个变化并保持连接处于打开状态，而非把所有可用的变化合并到一个 `results` 数组且随后关闭连接。在这种方式下，一旦发生变化，可以立刻返回更多 JSON 序列化的变化通知对象。

要想知道其工作原理，试试下面的命令（为了便于阅读，输出删减了）：

```
$ curl 'http://localhost:5984/music/_changes?since=97&feed=continuous'
{"seq":98,"id":"357999","changes":[{"rev":"1-0329f5c885...87b39beab0"}]}
{"seq":99,"id":"357998","changes":[{"rev":"1-79c3fd2fe6...1e45e4e35f"}]}
{"seq":100,"id":"357995","changes":[{"rev":"1-aa649aa53f...320c235aee"}]}
```

最后，如果一段时间内没有变化，CouchDB 会输下如下一行内容，然后关闭连接，

```
{"last_seq":100}
```

这种方式优于轮询与长轮询的地方在于，减少了开销，还让连接保持打开状态。于是，不会损失重建 HTTP 连接的时间。另一方面，输出不是单纯的 JSON，也就是说，解析要麻烦些。同理，如果客户端是 Web 浏览器，就会有问题。浏览器会异步地下载订阅，这会导致无法获取任何有效数据，直到整个连接完毕（这种情况下，使用长轮询更好）。

过滤变化

正如我们所见，Changes API 提供了一个独特的方式，让我们能够参与 CouchDB 数据库正在发生的事情。从好的方面看，它在单个数据流中提供了所有的变化。然而，有时你只想要变化的某个子集，而不是所发生的全部变化。比如，你可能只对文档删除感兴趣，或许，只关注某些特别的文档。这时，过滤器就起作用了。

过滤器有这样的功能，以某个文档（或者请求信息）作为输入，然后决定该文档是否应该通过过滤器。通过与否，都反映在返回值中。我们来看看这究竟是如何运作的。以 `music` 数据库为例，插入的大多数 artist 文档都有 `country` 属性，该属性包含一个三字母的编码。假设我们只对来自 Russia（RUS）的乐队感兴趣。过滤器看起来会像下面这样：

```
function(doc) {
  return doc.country === "RUS";
}
```

如果将这个函数加入“设计文档”的 `filters` 参数中，就能在发布`_changes` 请求的时候使用该函数。但是，在这么做之前，把例子做一番扩展。相比总想要 Russian 乐队，

更好的做法是将其参数化，这样 country 就能在 URL 里指定了。

这就是参数化 country 属性的过滤器函数：

```
function(doc, req) {
  return doc.country === req.query.country;
}
```

注意，如何比较文档的 `country` 属性与作为请求的查询串传入的同名属性。要看一个实例，可以只为基于地理位置的过滤器，新建一个“设计文档”并添加它：

```
$ curl -X PUT \
  http://localhost:5984/music/_design/wherabouts \
  -H "Content-Type: application/json" \
  -d '{"language":"javascript","filters":{"by_country":
    "function(doc,req){return doc.country === req.query.country;}"
  }}'
{
  "ok":true,
  "id":"_design/wherabouts",
  "rev":"1-c08b557d676ab861957eaeb85b628d74"
}
```

现在，可以发送一个过滤 country 变化的请求了：

```
$ curl "http://localhost:5984/music/_changes?\
filter=wherabouts/by_country&\
country=RUS"
{"results":[
{"seq":10,"id":"5987","changes":[{"rev":"1-2221be...a3b254"}]},
{"seq":57,"id":"349359","changes":[{"rev":"1-548bde...888a83"}]},
{"seq":73,"id":"364718","changes":[{"rev":"1-158d2e...5a7219"}]},
...
```

有了过滤器，你就能设置一种伪分片机制，其中，只有一部分记录在节点之间复制。尽管这不是像 MongoDB 或者 HBase 那样的真正分片系统，但它确实将处理某些请求的职责分离。比如，主 CouchDB 服务器可能有几个独立的过滤器，分别用于用户、订单、消息与库存。单独的 CouchDB 服务器能够基于这些过滤器复制变化，每个服务器支持业务的不同方面。

因为过滤器函数可以包含任意 JavaScript，所以可以纳入更复杂的逻辑。测试深层嵌套的域，与我们在创建视图时所做的类似。也能使用正则表达式测试属性或者进行数学比较（比如，根据日期范围过滤）。在请求对象中，甚至有 user context 属性（`req.userCtx`），

能用它在请求中找到类似用户名、密码这样的信息。

第 8 章再次讨论 Node.js 与 CouchDB 的 Changes API。而现在，我们会继续下去，讨论 CouchDB 最后一个出色特性：复制。

6.4.5　在 CouchDB 中复制数据

CouchDB 是一个关于异步环境与数据持久性的数据库。按照 CouchDB，存储数据最安全的地方无所不在，而 CouchDB 确实提供了这样的工具。我们看过的一些其他数据库，维护单个主节点以保证一致性。其他的某些数据库，则通过多数决议节点保证一致性。CouchDB 与两者都不同；它支持某种称为多主节点或者主—主的复制方式。

每个 CouchDB 服务器具有一样的能力，接收更新、响应请求以及删除数据，而这与它能否连接到其他服务器无关。在这种模型中，数据变化选择性地在一个方向上复制，所有的数据都以同一种方式复制。换句话说，没有分片。参与复制的服务器都将有全部数据。

复制是 CouchDB 中最后一个讨论的重要主题。首先会讨论如何在数据库间配置临时与连续的复制。然后，会解决数据冲突的影响，学习如何使应用程序以优雅的方式处理这些案例。

作为开始，点击页面右侧 Tools 菜单中的 Replicator 链接，这会打开一个如图 6-6 所示的页面。在 Replicate changes from 对话框，从左边的下拉菜单中选择 music，在右边输入 music-repl。取消勾选 Continuous 复选框，然后点击 Replicate 按钮。当出现提示时，点击 OK 按钮创建 `music-repl` 数据库。这应该在窗体下面的事件日志中生成一条事件消息。

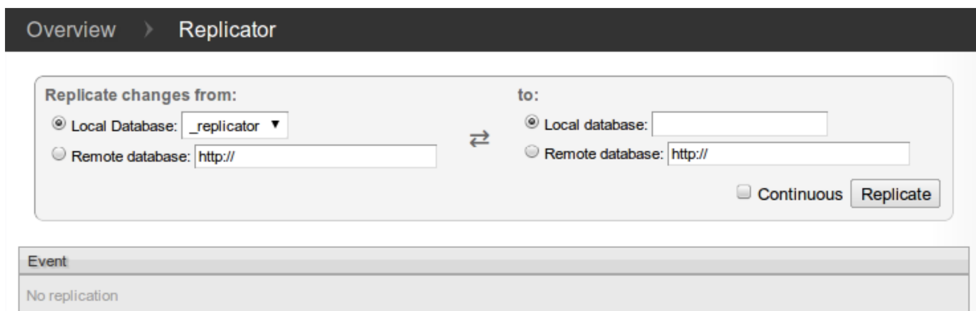


图 6-6　CouchDB Futon: Replicator

CouchDB 还是 BigCouch

CouchDB 的方式在许多用例中是合理且有效的，这无疑填补了我们讨论过的其他数据库所不能解决的缺口。另一方面，在节点之间选择性地复制数据是很棒的，这可以充分利用磁盘空间。也就是说，不是每个节点都包含全部数据，仅保存一定数量的副本。还记得 3.3.4 节讨论过的节点/写操作/读操作话题吗？这里的副本数，就是当时话题中 NWR 的 N。

CouchDB 不提供这样的特性，不过别担心！BigCouch 有这样的功能。Cloudant 开发并维护的 BigCouch，提供了兼容 CouchDB 的接口（仅有一些小的差别[a]）。但在接口之下，BigCouch 实现了分片与复制策略，类似于 Riak 这样的 Dynamo-inspired 数据库。安装 BigCouch 是一件挺繁琐的差事（比惬意的 CouchDB 难得多），但是，如果开发场景包含大型数据中心，这又是值得的。

[a] http://bigcouch.cloudant.com/api

为了确认复制请求生效了，回到 Futon Overview 页面。应该能看到一个名为 `music-repl` 的新数据库，其文档数目与 `music` 数据库相同。如果数目少了，等待些许时间并刷新页面——CouchDB 可能正在复制中。如果 Update Seq 的值不匹配，不要担心。这是因为原本的 `music` 数据库有文档的删除与更新操作，而 `music-repl` 数据库只有插入操作，以快速建立数据库。

1. 制造数据冲突

下一步，我们将制造数据冲突，然后探索解决冲突的办法。还是停留在 Replicator 页面上，因为我们打算在 `music` 与 `music-repl` 之间频繁地触发自由定义的复制。

回到命令行，输入以下内容，在 music 数据库里创建一个文档：

```
$ curl -X PUT "http://localhost:5984/music/theconflicts" \
  -H "Content-Type: application/json" \
  -d '{ "name": "The Conflicts" }'
{
  "ok":true,
  "id":"theconflicts",
  "rev":"1-e007498c59e95d23912be35545049174"
}
```

在 Replicator 页面上，点击 Replicate 按钮触发又一次同步。设法从 `music-repl` 数据库中检索该文档，验证同步是否成功完成。

```
$ curl "http://localhost:5984/music-repl/theconflicts"
{
```

```
  "_id":"theconflicts",
  "_rev":"1-e007498c59e95d23912be35545049174",
  "name":"The Conflicts"
}
```

然后，在 music-repl 中增加一张名为《Conflicts of Interest》的专辑，来更新该文档。

```
$ curl -X PUT "http://localhost:5984/music-repl/theconflicts" \
  -H "Content-Type: application/json" \
  -d '{
    "_id": "theconflicts",
    "_rev": "1-e007498c59e95d23912be35545049174",
    "name": "The Conflicts",
    "albums": ["Conflicts of Interest"]
  }'
{
  "ok":true,
  "id":"theconflicts",
  "rev":"2-0c969fbfa76eb7fcdf6412ef219fcac5"
}
```

为在 music 数据库中制造一次冲突更新，增加一张不同的专辑。

```
$ curl -X PUT "http://localhost:5984/music/theconflicts" \
  -H "Content-Type: application/json" \
  -d '{
    "_id": "theconflicts",
    "_rev": "1-e007498c59e95d23912be35545049174",
    "name": "The Conflicts",
    "albums": ["Conflicting Opinions"]
  }'
{
  "ok":true,
  "id":"theconflicts",
  "rev":"2-cab47bf4444a20d6a2d2204330fdce2a"
}
```

这时，music 和 music-repl 都有相同的_id 值 theconflicts。两个文档都是版本 2，且从一个基础版本（1-e007498c59e95d23912be35545049174）而来。现在的问题是，当我们试图复制二者时，会发生什么呢？

2. 解决数据冲突

在两个数据库之间，现在有相互冲突的两个文档。回到 Replicator 页面，再执行一次

复制。若你本以为复制会失败，很可能会震惊于操作竟然执行成功了。那么 CouchDB 究竟是如何处理数据差异的呢?

事实上，CouchDB 只是简单地二选一，将胜出的那个作为获胜者。当然，使用了确定的算法，所有的 CouchDB 节点都会在检测到冲突时，挑出同一个胜者。不过，事情还不止这些。CouchDB 也会存储未选中的“失败”文档，以便客户端应用能够在之后的某个时候回顾当时的情形，并解决问题。

要找出哪个版本的文档在最近的复制中“获胜”，可以用普通的 GET 请求获取该文档，通过增加 URL 参数 conflicts=true，CouchDB 也会将冲突的版本信息包含进来。

```
$ curl http://localhost:5984/music-repl/theconflicts?conflicts=true
{
  "_id":"theconflicts",
  "_rev":"2-cab47bf4444a20d6a2d2204330fdce2a",
  "name":"The Conflicts",
  "albums":["Conflicting Opinions"],
  "_conflicts":[
    "2-0c969fbfa76eb7fcdf6412ef219fcac5"
  ]
}
```

于是，我们看到第二次更新获胜了。注意响应中的`_conflicts`。它包含了与当前版本相冲突的其他版本列表。在 `GET` 请求中增加 rev 参数，可以获取到所有冲突的修订版本，并决定如何处置它们。

```
$ curl http://localhost:5984/music-repl/theconflicts?rev=2-0c969f...
{
  "_id":"theconflicts",
  "_rev":"2-0c969fbfa76eb7fcdf6412ef219fcac5",
  "name":"The Conflicts",
  "albums":["Conflicts of Interest"]
}
```

这里的启发在于 CouchDB 没有设法智能地合并冲突的数据变化。至于如何合并冲突的文档，这与特定应用相关，很难找到一个普遍适用的解决方案。在我们的例子中，通过连接两个 albums 数组的方法实现合并是有意义的，但你很容易会想到在一些情况下，合适的操作并非显而易见。

举例来说，假设你正在维护一个关于日历事件的数据库。你的智能手机上有一个副本；笔记本电脑上有另一个副本。现在，你收到派对策划者发来的一条短消息说，你委

托举办派对的地点确定了。于是，你更新了智能手机上的日历数据库。稍后，回到办公室，你收到派对策划者发来的一封邮件，其中提到另一个派对地点，所以，你更新笔记本上的数据库，然后同步智能手机与笔记本上的数据库。CouchDB 无法知道哪个派对地点是正确的。此时，最好的做法是保持一致，保存旧值，以便你能够核查究竟哪个冲突值应该保留。由应用程序选择合适的用户接口，处理这种情况，并征求一个决策——不失为一种好的方式。

6.4.6　第 3 天总结

我们的 CouchDB 之旅到此结束了。在第 3 天里，我们学习了如何为映射规约产生的视图增加规约函数。然后，我们深入了解了 Changes API，包括如何开发基于 Node.js 的事件驱动的服务器端 JavaScript。最后，我们了解了 CouchDB 是如何实现主-主复制策略的，以及客户端应用是怎样侦测并解决冲突的。

第 3 天作业

求索

1. CouchDB 中有什么可用的原生规约器吗？如果有的话，相比使用定制的 JavaScript 规约器，又有什么好处？
2. 在服务器端，如何过滤出自`_changes API`的数据变化呢？
3. 与 CouchDB 中的所有任务一样，初始化与取消复制的任务实际也是由 HTTP 命令控制的。在服务器之间设置与移除复制关系的 REST 命令又是什么？
4. 如何使用`_replicator`数据库，保存复制关系？

实践

1. 在 Node.js 模块（6.4.3 节讨论过）的基础上，创建名为`watch_changes_continuous.js`的新模块。
2. 实现`watcher.start()`，使其以连续的方式监控`_changes`订阅内容。确认它与`watch_changes_longpolling.js`产生相同的输出。

提示：如果你遇到问题，可以下载本书的示例实现，作为参考。

3. 带有冲突版本的文档有一个`_conflicts`属性。创建一个视图，发布冲突的版本，并将它们映射到`doc_id`。

6.5 总结

通过本章的学习，我们见识了如何执行与 CouchDB 相关的很多任务，从基本的 CRUD 操作，到利用映射规约函数创建视图。我们学习了如何监视数据变化，还尝试开发了非阻塞的事件驱动客户端应用。最后，我们讨论了如何在数据库之间执行自由定义的复制，以及侦测并解决冲突。除了所有这些，其实还有很多我们没有接触的内容。不过，在开始讨论下一种数据库之前，总结一下所学是必要的。

6.5.1 CouchDB 的优点

CouchDB 是 NoSQL 社区中，健壮且稳定的一员。网络是不可靠的，而硬件故障总是迫在眉睫——CouchDB 就是基于这种哲学建立的，于是，提供了一种尽可能分散的数据存储方式。小，小到可以运行在智能手机中；大，大到足以支持企业应用。CouchDB 能够胜任各种部署场景。

CouchDB 是一个作为数据库的 API。在本章中，我们关注 canonical Apache CouchDB 项目，但事实上，有越来越多的可选实现，而 CouchDB 服务提供者则建立于混合的后端之上。因为 CouchDB 由 Web 起，为 Web 生，应用网络技术是相当自然的——比如负载均衡器与缓存层——当然，最终都是通过 CouchDB 的 API 可用的技术。

6.5.2 CouchDB 的缺点

当然，CouchDB 不是万能的。CouchDB 中基于映射规约的视图，虽然新颖，但是不能执行关系数据库中的数据分片。事实上，在生产环境中，你不应该运行自由定义的查询。同样，CouchDB 的复制策略不总是正确的选择。CouchDB 的复制是全部或者全无，也就是说，所有参与复制的服务器有着一样内容。于是，在数据分布到数据中心的过程中，没有分片。增加 CouchDB 节点的主要理由，不是为了将数据四处分散，而是尽可能增加读写操作的吞吐量。

6.5.3 结束语

为处理不确定性 CouchDB，尤其关注健壮性；因此，如果你的系统必须运行于残酷的互联网环境中，CouchDB 会是很棒的选择。通过利用标准的 Web 方法，例如

HTTP/REST 与 JSON，CouchDB 总是能轻易适应流行的 Web 技术。在数据中心的围墙内，由 CouchDB 或 BigCouch 应对数据冲突，这都没有问题；但是不要奢望数据分片。

还有很多其他我们尚未提到的特性，使 CouchDB 成为独一无二的数据库。这些特性大致包括易于备份、文档支持二进制附件以及 CouchApps——不通过中间件，直接开发并部署 Web 应用的系统。说了这么多，我们希望给了你足够的亮点，吊起你的胃口，自己继续学习下去。若要选择一个数据驱动的 Web 应用，不妨试试 CouchDB；你不会失望的！

第 7 章

Neo4j

蹦极用的绳索看上去并不像一件标准的木匠工具，正如Neo4j似乎并不像标准的数据库。蹦极绳不只是蹦极绳，它可以用来绑东西——奇形怪状的东西皆可。如果把桌子[1]、柱子和运货的一辆皮卡绑在一起对你来说很重要，听我的，随身带着蹦极绳吧。

Neo4j 是一种新型的 NoSQL 数据存储，称为图数据库。顾名思义，Neo4j 将数据另存为图（数学意义上的图形）。同时，它也称为“白板友好”的数据库，也就是说，如果能在白板上设计一些框和线条，就可以用 Neo4j 把它存起来。Neo4j 的重点是数据间的关系，而非数据集合间的共性（比如，文档集合或者数据行组成的表）。在这种方式下，Neo4j 能以自然而直接的方式存储多变的数据。

Neo4j 很小，小到足以嵌入几乎任何应用程序。另一方面，它能够存储数百亿的节点与相同数量的边。并且，一旦有了集群支持，在多个服务器间实现主从副本，Neo4j 几乎能处理任何规模的问题。

7.1 Neo4j，白板友好的数据库

设想一下，你要创建一个葡萄酒建议引擎，众所周知，葡萄酒有不同的品种、产区、酒庄、年份以及标志。此外，你可能得记录描述葡萄酒的文章。也许，你还想让用户能够跟踪他们最喜爱葡萄酒的信息。

对于这样的需求，关系模型会创建一个目录（category）表，以及某个酒庄的酒与多个目录或者一些其他数据的一种多对多关系。可是，这与人类建模数据的方式不太一样。比较

[1] 译者注：这里的 table 和 column 一语双关，在数据库中，它们分别指表和列。

这两张图：图 7-1 与图 7-2。关系数据库领域有句老话：只要时间够长，所有的字段都不是必需的。Neo4j 只在必要之处提供数据值与结构，从而默默地解决了这个问题。如果一种调制葡萄酒没有年份，取而代之的是装瓶年份，以表明调制时间。而模式无法为此做任何调整。

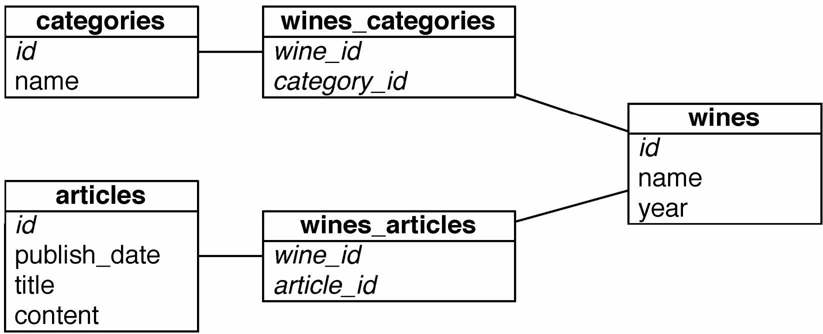


图 7-1　关系 UML 描述的模式



图 7-2　白板上的葡萄酒建议数据

在未来的三天里，我们将学习如何通过控制台、REST 以及搜索索引与 Neo4j 交互。我们也会根据图算法，驾驭规模更大的图形。最后，在第 3 天，我们将概览 Neo4j 为关键任务提供的企业级工具，其范围包括：完全遵循 ACID 的事务、高可用性的集群以及增量备份。

本章会使用 Neo4j 1.7 企业版。尽管 GPL 社区版能支持我们所需的大部分操作，但是，第 3 天我们需要一些企业级的功能：高可用性。

7.2　第 1 天：图、Groovy 和 CRUD

读者朋友请不要意外，第一天我们就不打算浅尝辄止。除了探究 Neo4j 的 Web 接口，

我们还将深入探索图数据库的术语以及 CRUD（Create/Read/Update/Delete）。今天的主要内容是学习如何通过一个称为*遍历*（walking）的过程，查询图数据库。在本章中出现的概念，与我们到目前为止已经看到的其他数据库相比，差别很大。这些数据库主要以文档或者基于记录的形式展现这个世界。而在 Neo4j 中，一切都与关系（relationship）有关。

不过，在我们接触这些内容之前，不妨以 Web 接口为起点，看看 Neo4j 是如何以图的形式表示数据，以及 Neo4j 是如何遍历图的。首先下载并解压缩 Neo4j 包，然后用 `cd` 命令进入目录并以如下命令启动服务器：

```
$ bin/neo4j start
```

为确保服务器启动并开始运行，用 curl 命令访问如下 URL：

```
$ curl http://localhost:7474/db/data/
```

与 CouchDB 一样，默认的 Neo4j 包自带一个相当好用的 Web 管理工具以及数据浏览器，这个浏览器尤其适合运行一些简单命令。如果这还不够，Neo4j 配备了我们见过的最酷的图数据浏览器之一。当然，自带的工具足以提供一个完美的开始，这是因为第一次接触图遍历会觉得很别扭，工具愈简单愈好。

7.2.1　Neo4j之Web接口

打开 Web 浏览器，访问其管理页面。

```
http://localhost:7474/webadmin/
```

你会看到一幅彩色但依旧为空的图，如图 7-3 所示。点击页面顶端的 Data browser（数据浏览器）选项。你会发现新安装的 Neo4j 有一个事先存在的节点：node 0。

图数据库中的*节点*与前面章节谈到的节点并非完全没有关联。之前，我们提到节点，意思是网络中的物理服务器。你若把整个网络看做一幅巨大的相互连接的图，一个服务器节点就是一个点或者*顶点*，而服务器之间的关系，就是边。

在 Neo4j 中，节点的概念与之类似：作为边之间的顶点，能够以键值（key-value）集合的形式存放数据。点击+Property 按钮，将 name 作为键，Prancing Wolf Ice Wine 2007 作为值，表示某种葡萄酒及其年份。然后，点击+Node 按钮，如图 7-4 所示。在新建的节点上，增加 `name` 属性，其值设置为 Wine Expert Monthly（我们将它简写为：`[name : "Wine Expert Monthly"]`）。不难发现，节点的编号是自动增加的。

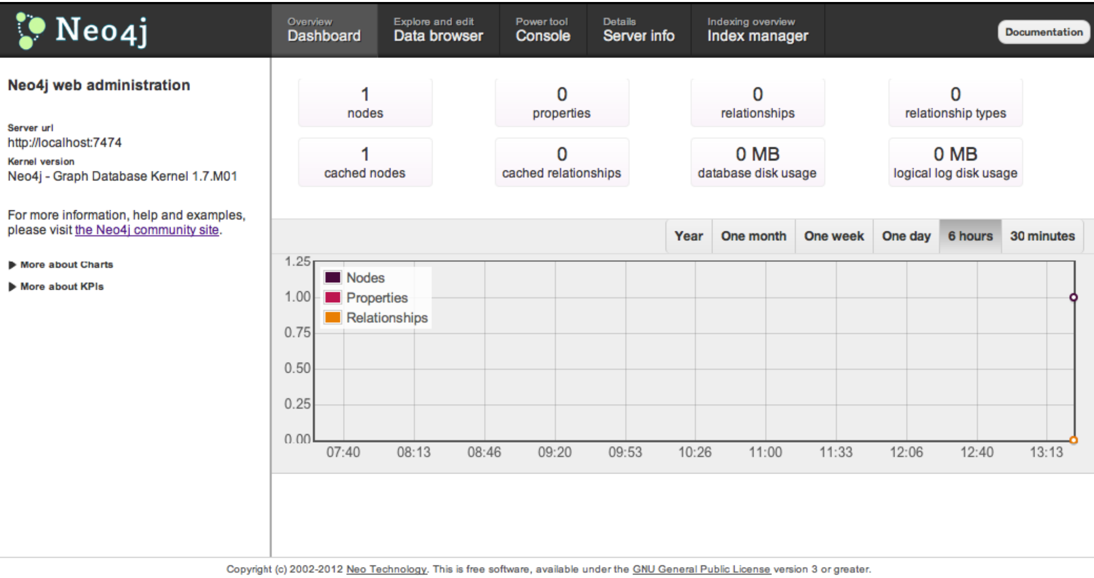


图 7-3　Web 管理页面仪表板



图 7-4　点击+Node 按钮添加新节点

现在，我们有两个节点，但它们之间没有关联。由于 Wine Expert 报道了葡萄酒 Prancing Wolf，因此得通过创建边，将这两个节点联系起来。点击+Relationship 按钮，从 node 1 指向 node 0，关系类型为 `reported_on`。

对于这个新建的关系，可以通过如下 URL 查看其详细信息...

```
http://localhost:7474/db/data/relationship/0
```

该关系表明 Node 1 报道了 Node 0。

与节点一样，关系也能包含属性。点击+ Add Property 按钮，输入属性`[rating : 92]`，以此记录该葡萄酒的评分。

这种特殊的冰葡萄酒由雷司令葡萄（riesling grape）[1]，酿制而成，我们得把这条信息加入数据库。一种方法是直接在表示该葡萄酒的节点上添加属性，但是雷司令其实是一种类别，其他葡萄酒也会划入此类，因此更合适的方法是创建一个节点，并将其属性设置为`[name : "riesling"]`。然后，添加从node 0 指向node 2 的关系`grape_type`，并设置属性`[style :`

[1] 一种起源于德国莱茵地区的葡萄。——译者注

"ice wine"]。

对了，我们的图看起来会是什么样子呢？点击"switch view mode"按钮（在+Relationship 按钮旁边，看起来歪歪扭扭的那个），你看到的画面，类似图 7-5。

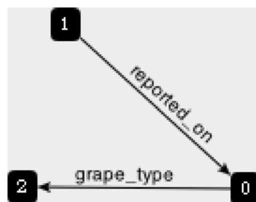


图 7-5 与当前节点相关的节点图

点击 Style 按钮，弹出一个菜单，可以从中选择配置，用于渲染图形的可视化效果。想在图上看到更多的有用信息，先点击 Style 按钮，然后 New Profile 按钮，你会看到页面"Create new visualization profile"。在页面顶部，输入 wines 作为配置名，再将 label 输入框的值从`{id}`改成`{id}: {prop.name}`。点击 Save 按钮，回到可视化页面。现在，可以从 Style 菜单选择 wines 配置，页面渲染效果类似图 7-6。

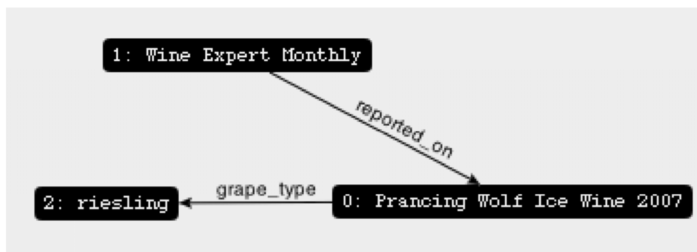


图 7-6 自定义配置的节点图

通过 Web 接口进行编辑是一种简单易学的方法，但是我们还是需要更强大的接口以应对实际的生产工作。

7.2.2 通过Gremlin操作Neo4j

有若干种编程语言能与Neo4j进行互操作：Java代码、REST、Cypher以及Ruby控制台（Ruby console）等。我们今天会用一种名为Gremlin的编程语言，它是用Groovy编程语言实现的图遍历语言。然而，使用Gremlin事实上不需要了解Groovy，所以不妨将它看做另一种声明式[1]的领域特定语言，好比SQL。

---

[1] 关于 declarative programming 的详细信息参见 http://en.wikipedia.org/wiki/Declarative_programming。——译者注

与我们研究过的其他控制台一样，Gremlin 提供了访问其底层语言架构的能力。也就是说，可以在 Gremlin 中使用 Groovy 构造函数与 Java 类库。我们发现，较之 Neo4j 的原生 Java 代码，Gremlin 与图交互的方式更强大也更自然。更厉害的是，Gremlin 控制台可以在 Web 管理工具中使用；只须点击 Web 管理页面顶部的 Console 链接，并选择 Gremlin。

方便起见，以变量 g 表示图对象。对图的操作即是作用于变量 g 的函数。

由于 Gremlin 是通用的图遍历语言，它使用数学中的通用图术语。Neo4j 将存储数据的图形点称为节点（node），Gremlin 则称为顶点（vertex）；Neo4j 中的关系（relationship），Gremlin 称为边（edge）。

为访问图中所有的顶点，以名为 V 的属性表示全部顶点，输入如下命令访问它们。

```
gremlin> g.V
==>v[0]
==>v[1]
==>v[2]
```

类似地，还有名为 E 的姊妹属性表示所有的边。

```
gremlin> g.E
==> e[0][1-reported_on->0]
==> e[1][0-grape_type->2]
```

可以在 v（小写）方法中传入节点号，访问某个特定的顶点。

```
gremlin> g.v(0)
==> v[0]
```

为了确保访问正确的顶点，可以通过 map() 方法，把顶点的属性罗列出来。注意，可以在 Groovy/Gremlin 中使用方法调用链，如下所示。

```
gremlin> g.v(0).map()
==> name=Prancing Wolf Ice Wine 2007
```

v（0）能准确返回想要的节点，不过，你也可以在所有节点中过滤出你想要的某个值。比如，按名字检索雷司令（riesling），可以使用过滤器语法{…}，这在 Groovy 代码中称为闭包（closure）。花括号{…}所包围的全部代码定义了函数，如果某个顶点使该函数计算结果为 true，则这个顶点会过滤出。闭包中的关键字 it 表示当前遍历的对象，它会自动填充合适的值，以供你使用。

```
gremlin> g.V.filter{it.name=='riesling'}
==> v[2]
```

一旦取到某个顶点，就可以对该顶点调用 `outE()`，从而得到从这个顶点出发的所有边。由 inE()获取进入的边，由 bothE()调用进入和出发的边。

```
gremlin> g.V.filter{it.name=='Wine Expert Monthly'}.outE()
==> e[0][1-reported_on->0]
```

注意，和 Ruby 一样，在 Groovy 中，方法的圆括号是可选的，所以调用 outE 也行。

```
gremlin> g.V.filter{it.name=='Wine Expert Monthly'}.outE
==> e[0][1-reported_on->0]
```

基于向外的边，可以通过 `inV` 方法，得到这些边所指向的顶点。边 `reported_on` 由顶点 Wine Expert 指向顶点 Prancing Wolf Ice Wine 2007，所以 `outE.inV` 会返回顶点 Prancing Wolf Ice Wine 2007。然后，检索该顶点的 `name` 属性，语法如下。

```
gremlin> g.V.filter{it.name=='Wine Expert Monthly'}.outE.inV.name
==> Prancing Wolf Ice Wine 2007
```

表达式 `outE.inV` 会寻找输入顶点（在下例中即 Wine Expert Monthly）通过边所指向的顶点。而相反的操作（寻找这样的顶点，它们有指向输入顶点的边）可由 `inE.outV` 完成。因为这对操作相当常用，所以 Gremlin 提供了它们的快捷方式。表达式 `out` 是 `outE.inV` 的缩写，`in` 是 `inE.outV` 的缩写。

```
gremlin> g.V.filter{it.name=='Wine Expert Monthly'}.out.name
==> Prancing Wolf Ice Wine 2007
```

一个酒庄会酿造多种葡萄酒，所以，如果我们计划添加更多的葡萄酒，就得为酒庄新建一个节点，并新建一条指向 Prancing Wolf 的边。

```
gremlin> pwolf = g.addVertex([name : 'Prancing Wolf Winery'])
==> v[3]
gremlin> g.addEdge(pwolf, g.v(0), 'produced')
==> e[2][3-produced->0]
```

由下面的代码添加两种雷司令葡萄酒：Kabinett 和 Spatlese。

```
gremlin> kabinett = g.addVertex([name : 'Prancing Wolf Kabinett 2002'])
==> v[4]
gremlin> g.addEdge(pwolf, kabinett, 'produced')
==> e[3][3-produced->4]

gremlin> spatlese = g.addVertex([name : 'Prancing Wolf Spatlese 2007'])
==> v[5]
```

```
gremlin> g.addEdge(pwolf, spatlese, 'produced')
==> e[4][3-produced->5]
```

让我们添加从 riesling 顶点到新增顶点的几条边，完成这张小图。通过滤出 riesling 顶点，为变量 riesling 赋值；然后，需要 `next()` 方法，获取 pipeline（管道）的第一个顶点。稍后就会详细了解 pipeline 的概念。

```
gremlin> riesling = g.V.filter{it.name=='riesling'}.next()
==> v[2]
gremlin> g.addEdge([style:'kabinett'], kabinett, riesling, 'grape_type')
==> e[5][4-grape_type->2]
```

用类似的方法，建立从 Spatlese 指向 riesling 的边，不同的是 `style` 设置为 `spatlese`。当这些数据添加完毕时，该图在可视化工具里如图 7-7 所示。

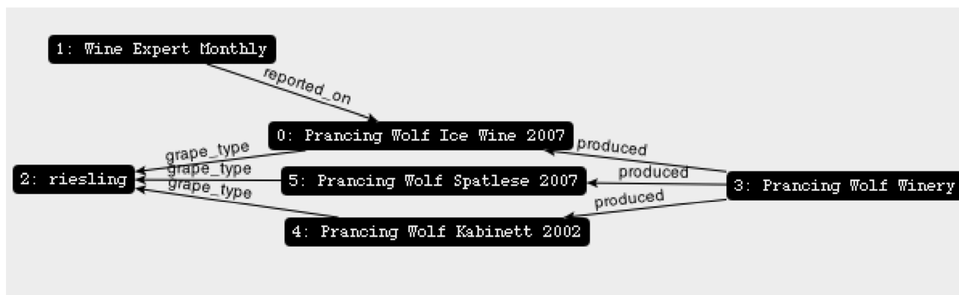


图 7-7　由 Gremlin 添加数据后的节点图示

7.2.3　pipe的威力

可以把 Gremlin 操做看做一系列 pipe。每个 pipe 一边以某个数据集作为输入，一边吐出数据集作为输出。一个数据集可以包含一个数据项，多个数据项，或者没有数据项。而数据项可以是顶点、边或者属性值。

比如，`outE` pipe吸入顶点的数据集，吐出边的数据集。一系列的pipe称为pipeline，它以声明方式描述问题。对比典型的命令式编程（imperative programming）[1]方法，pipe要求描述每个步骤从而解决问题。事实上，pipe是查询图数据库最简洁的方法之一。

[1] 关于 imperative programming 的更多信息参见 http://en.wikipedia.org/wiki/Imperative_programming。——译者注



Jim 谈：jQuery 与 Gremlin

jQuery JavaScript 库十分流行，它的用户或许会发现 Gremin 的面向集合的遍历方法与之相当类似。考虑如下 HTML 片段:

```
<ul id="navigation">
  <li>
    <a name="section1">section 1</a>
  </li>
  <li>
    <a name="section2">section 2</a>
  </li>
</ul>
```

现在，假定我们想查找名为 section1 的所有 tag 中的文本，即在 navigation 元素（id=navigation）之下的 list 项（<li>）的子项。

用 jQuery 进行这个查询的方法之一是使用如下代码:

```
$('[id=navigation]').children('li').children('[name=section1]').text()
```

接着，看看 Gremin 查询是如何达成类似功能的。不妨想象每个父节点都有指向每个子节点的边:

```
g.V.filter{it.id=='navigation'}.out.filter{it.tag=='li'}.
out.filter{it.name=='section1'}.text
```

是不是很像，嗯？

从本质上说，Gremlin 是一种用来建立 pipe 的语言。具体而言，它是以称为 Pipes 的 Java 项目为基础的。

为探讨 pipe 的概念，我们回到先前的葡萄酒图。假设我们想查找与给定葡萄酒类似的酒——也就是，它们种类相同。我们可以从一种冰葡萄酒开始，该酒与其他 out 节点有相同 grape_type 类型的边（这里忽略初始葡萄酒节点 v(0)）。

```
ice_wine = g.v(0)
ice_wine.out('grape_type').in('grape_type').filter{ !it.equals(ice_wine) }
```

如果你曾在工作中使用 Smalltalk 或者 Rails，一定对这种风格的方法链似曾相识。但是，对比这种形式的方法链与之后，我们会看到标准的 Neo4j Java API，在后者中，一个节点的关系必须依次迭代，以访问各个节点。

```
enum WineRelationshipType implements RelationshipType {
  grape_type
}
import static WineRelationshipType.grape_type;
public static List<Node> same_variety( Node wine ) {
  List<Node> wine_list = new ArrayList<Node>();
  // walk into all out edges from this vertex
  for( Relationship outE : wine.getRelationships( grape_type ) ) {
    // walk into all in edges from this edge's out vertex
    for( Edge inE : outE.getEndNode().getRelationships( grape_type ) ) {
      // only add vertices that are not the given vertex
      if( !inE.getStartNode().equals( wine ) ) {
        wine_list.add( inE.getStartNode() );
      }
    }
  }
  return wine_list;
}
```

不同于上面所示的嵌套与迭代，Pipes 项目设计了一种方法，可以声明出发（outgoing）与进入（incoming）顶点。可以新建一系列进出 pipe、过滤器，并从 pipeline 中取值。然后迭代地调用 pipeline 的 `hasNext()`方法，该方法会返回下一个匹配的节点。换句话说，pipeline 会自动遍历整棵树。只需声明如何遍历，pipeline 会按要求查询需要的值。

为了加以说明，下面是 `same_variety()`方法的另一种实现，其中使用了 Pipes 而非显式地循环：

```
public static void same_variety( Vertex wine ) {
  List<Vertex> wine_list = new ArrayList<Vertex>();
  Pipe inE = new InPipe( "grape_type" );
  Pipe outE = new OutPipe( "grape_type" );
  Pipe not_wine = new ObjectFilterPipe( wine, true );
  Pipe<Vertex,Vertex> pipeline =
    new Pipeline<Vertex,Vertex>( outE, inE, not_wine );
  pipeline.setStarts( Arrays.asList( wine ) );
  while( pipeline.hasNext() ) {
      wine_list.add( pipeline.next() );
  }
  return wine_list;
}
```

从深处来说，Gremlin 是由 `Pipe` 建立的语言。遍历图的工作，当然还是由 Neo4j 服务器执行，因此需要建立 Neo4j 能够理解的查询，而 Gremlin 着实简化了这项工作。

7.2.4　Pipeline与顶点

为了抓取只包含某个特定顶点的集合，可以从所有节点的列表中将它过滤出。比如，调用 g.V.filter{it.name=='reisling'}，就是这种方式。属性 V 是所有节点的列表，可以从中拣选一个子列表。但是，当我们想要顶点本身，而非一个列表时，就需要调用 next() 方法。该方法检索 pipeline 的第一个顶点。简而言之，我们所需要区别的，类似于只包含一个元素的数组与这个元素本身之差别。

如果查看过滤器 filter 的 class 属性，请注意，它返回的是类型 GremlinPipeline。

```
gremlin> g.V.filter{it.name=='Prancing Wolf Winery'}.class
==>class com.tinkerpop.gremlin.pipes.GremlinPipeline
```

将它与 pipeline 的 next()方法所返回的类型相比，后者返回的是不同的类型，即 Neo4jVertex。

```
gremlin> g.V.filter{it.name=='Prancing Wolf Winery'}.next().class
==>class com.tinkerpop.blueprints.pgm.impls.neo4j.Neo4jVertex
```

虽然控制台可以方便地列出从 pipeline 中检索到的节点，但是 pipeline 终究是 pipeline，除非显式地从中取出什么内容。

7.2.5　无模式的社会性数据

在图中建立社会性数据，可以通过添加更多节点，简单地实现。假设我们想添加三个人物——两个人互相认识，第三个则是陌生人，每个人都有自己偏爱的酒。

Alice 有点嗜甜，因此热衷冰葡萄酒。

```
alice = g.addVertex([name:'Alice'])
ice_wine = g.V.filter{it.name=='Prancing Wolf Ice Wine 2007'}.next()
g.addEdge(alice, ice_wine, 'likes')
```

Tom 喜欢 Kabinett 和冰葡萄酒，并相信《Wine Expert Monthly》发表的任何文章。

```
tom = g.addVertex([name:'Tom'])
kabinett = g.V.filter{it.name=='Prancing Wolf Kabinett 2002'}.next()
g.addEdge(tom, kabinett, 'likes')
g.addEdge(tom, ice_wine, 'likes')
g.addEdge(tom, g.V.filter{it.name=='Wine Expert Monthly'}.next(), 'trusts')
```

Patty 是 Tom 和 Alice 的朋友，但是，Patty 才接触葡萄酒不久，还未选择任何钟爱的酒。

```
patty = g.addVertex([name:'Patty'])
g.addEdge(patty, tom, 'friends')
g.addEdge(patty, alice, 'friends')
```

我们能够超出设计原意，添加新的行为，却得以避免对已有图的基本结构做任何修改。新增的节点是相互关联的，如图 7-8 所示。

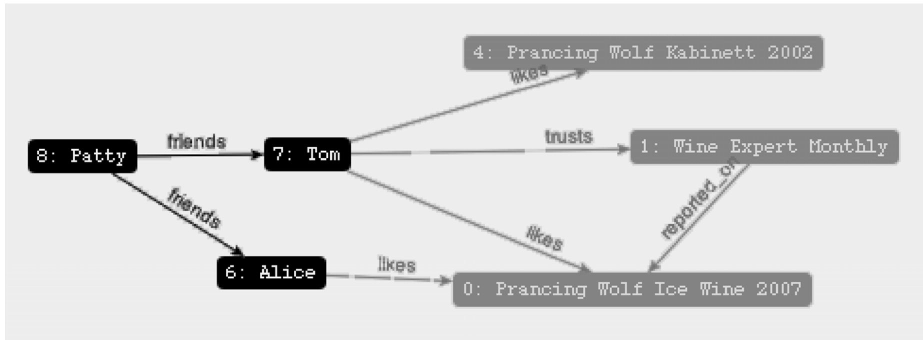


图 7-8　表示社会性数据的节点子集



Eric 谈：Cypher 语言

Cypher 是 Neo4j 支持的另一种图查询语言，它基于模式匹配（pattern matching），且语法类似 SQL。Cypher 语句让人感到十分熟悉，容易理解。尤其是，它的 MATCH 语句十分直观，是一种类似 ASCII 艺术的表达式。

一开始，我嫌 Cypher 过于冗长，但是，随着我的眼睛一次次适应 Cypher 的语法，我成了它的支持者。

前文讨论的“类似葡萄酒”查询，在 Cypher 中以如下方式实现:

```
START ice_wine=node(0)
MATCH (ice_wine) -[:grape_type]-> () <-[:grape_type]- (similar)
RETURN similar
```

我们先将 ice_wine 绑定到 node 0。MATCH 语句使用圆括号中的标识符表示节点，诸如-[:grape_type]->之类的箭头表示有向的关系。事实上，我喜欢这种书写查询语句的方式，原因在于我们轻而易举地形象化节点的遍历。

而且，你可以很快地从入门到进阶。下面是一个更实际的例子——每一处都与 SQL 一样强大，易于理解。

```
START ice_wine=node:wines(name="Prancing Wolf Ice Wine 2007")
MATCH ice_wine -[:grape_type]-> wine_type <-[:grape_type]- similar
WHERE wine_type =~ /(?i)riesl.*)/
RETURN wine_type.name, collect(similar) as wines, count(*) as wine_count
ORDER BY wine_count desc
LIMIT 10
```

尽管我在本章节主要选择 Gremlin 进行讨论，实际上两种语言互为补充，并能和谐共存。在日常工作中，你会根据思考问题的不同角度，选择合适的语言。

7.2.6 垫脚石

我们已经看了一些基本的 Gremlin 步骤，或者 Pipe 处理单元。而 Gremlin 能做的，要比这些多得多。接着，我们会讨论更多构件，不仅仅是图遍历，还包括对象转换、过滤以及附加的功能，比如统计按条件（criteria）分组的节点。

我们已经看了 inE、outE、inV 和 outV，它们都是检索进出边与顶点的转换步骤。还有其他两个类型 bothE 与 bothV，它们就循着边，不管边的方向是进还是出。

下面的语句检索 Alice 和她所有的朋友。我们把 name 放在语句的最后，以获取顶点的 name 属性。由于我们不在意 friend 边的方向，因此在此用的是 bothE 和 bothV。

```
alice.bothE('friends').bothV.name
==> Alice
==> Patty
```

如果你不希望 Alice 出现在结果中，可以将我们不想要的节点列表，传入 except()过滤器，except()才会遍历剩下的节点。

```
alice.bothE('friends').bothV.except([alice]).name
==> Patty
```

与 except()功能相反的是 retain()，如你所猜测，retain()只遍历匹配的节点。

另一种替代方法是用代码块过滤返回的顶点，在该代码块中，会检查当前顶点是否等于 alice。

```
alice.bothE('friends').bothV.filter{!it.equals(alice)}.name
```

如果你想认识 Alice 的朋友的朋友，又该怎么办呢？你只要重复上面提到的步骤即可，如下所示：

```
alice.bothE('friends').bothV.except([alice]).
bothE('friends').bothV.except([alice])
```

用同样的方法，可以查询 Alice 的朋友的朋友的朋友，只需在方法链后再加一组 bothE/bothV/except 调用即可。但是，我们会为此录入很多字，并且不可能以这种方式书写 *n* 步（*n* 是表示 friends 关系层数的变量）。而 loop()方法正是为此设计的，它会重复若干次之前的步骤，只要闭包中的给定条件为真，就会继续执行。

下面的代码会将 loop()之前的三步进行循环，这可以通过从 loop 调用的前数三个“.”来实现。于是，except 是一步，bothV 是两步，bothE 是三步。

```
alice.bothE('friends').bothV.except([alice]).loop(3){
  it.loops <= 2
}.name
```

每次执行大量循环步骤后，loop()会调用闭包中的给定条件——花括号{...}之间的代码。其中，it.loops 属性会记录当前循环执行的次数。在这个例子中，我们检查循环次数是否小于等于 2，也就是说，循环执行两次后就会停止。事实上，这里的闭包很像普通编程语言中的 while 语句。

```
==>Tom
==>Patty
==>Patty
```

loop 的确可行，正确地找到了 Tom 和 Patty。但是，不难发现，Patty 出现了两次。这是因为，Patty 一次是作为 Alice 的朋友，另一次是作为 Tom 的朋友匹配的。所以，我们需要过滤掉重复对象的方法，这正是 dedup()过滤器提供的功能。

```
alice.bothE('friends').bothV.except([alice]).loop(3){
  it.loops <= 2
}.dedup.name

==>Tom
==>Patty
```

为了深入了解获取这些值的路径，可以使用 paths()转换，跟踪 friend->friend 的路径。

```
alice.bothE('friends').bothV.except([alice]).loop(3){
  it.loops <= 2
}.dedup.name.paths

==> [v[7], e[12][9-friends->7], v[9], e[11][9-friends->8], v[8], Tom]
==> [v[7], e[12][9-friends->7], v[9], e[11][9-friends->8], v[9], Patty]
```

到目前为止，所做的遍历都是循着图向前进行的。有时，需要前进两步，后退两步。从 Alice 节点开始，前进两步，后退两步，正好回到 Alice 节点。

```
gremlin> alice.outE.inV.back(2).name
==> Alice
```

我们探讨的最后一个常用的步骤是 `groupCount()`，它会遍历节点，对重复值计数，并将它们抓取在一个 map 中。

考虑下面的例子，获取图中所有顶点的 `name` 属性，统计每个 name 分别出现几次：

```
gremlin> name_map = [:]
gremlin> g.V.name.groupCount( name_map )
gremlin> name_map
==> Prancing Wolf Ice Wine 2007=1
==> Wine Expert Monthly=1
==> riesling=1
==> Prancing Wolf Winery=1
==> Prancing Wolf Kabinett 2002=1
==> Prancing Wolf Spatlese 2007=1
==> Alice=1
==> Tom=1
==> Patty=1
```

在 Groovy/Gremlin 中，map 由命名规则`[:]`表示，与 Ruby/JavaScript 中的记法`{}`十分相像。注意，map 中所有的值都是 1。这跟我们预期的一样，因为并没有任何重复的 name，而 `V` 集合正好包含图中每个节点的一个副本。

接下来，统计每个人喜爱的酒的数量。我们可以得到每个人喜爱的顶点，对它们按照 name 计数即可。

```
gremlin> wines_count = [:]
gremlin> g.V.outE('likes').outV.name.groupCount( wines_count )
gremlin> wines_count
==> Alice=1
==> Tom=2
```

如我们所预期，Alice 喜爱一种酒，Tom 喜爱两种酒。

7.2.7 引入Groovy

除了 Gremlin 步骤，我们也有 Groovy 语言的各种构造器和方法。Groovy 有名为`collect()`的映射（map）函数（mapreduce 风格的）以及名为`inject()`的规约（reduce）

函数。由此，我们可以使用类似 mapreduce 的查询。

考虑这样的案例，我们想统计尚无评价的葡萄酒的数量。我们可以先通过映射，得到一个表示是否评价过的 true/false 值列表，然后将这个列表规约，对所有的 true/false 计数。映射部分用到 collect 函数，如下所示：

```
rated_list = g.V.in('grape_type').collect{
  !it.inE('reported_on').toList().isEmpty()
}
```

在上面的代码中，表达式 g.V.in（'grape_type'）即 g.V.inE（'grape_type'）.outV，返回全部有向外关系 grape_type 的节点。只有 wine 节点才有这样的边，因此我们得到了系统中全部 wine 节点的列表。然后，在 collect 闭包中，我们会检查 wine 节点是否有进边（incoming edge）reported_on。toList()会将某 wine 顶点的所有 reported_on 边变成一个列表，这样我们就能判断这个列表是否为空，从而决定某个 wine 顶点是否评价过。

为统计多少酒尚未评价，只需用 inject()方法充当规约器（reducer）。

```
rated_list.inject(0){ count, is_rated ->
  if (is_rated) {
    count
  } else {
    count + 1
  }
}
==> 2
```

在 Groovy 中，箭头运算符（->）将闭包的输入参数与闭包体隔开。在这个规约器中，要记录累计值，还要判断当前处理的葡萄酒是否评价过，所以这里有 count 和 is_rated。inject(0)中的 0 将 count 初始化为 0，然后在闭包体内，对评价过的葡萄酒返回 count 的当前值，对未评价过的返回 count+1。最终的输出，就是列表中 false 值的总数（也就是，未评价的酒的数量）。

```
==> 2
```

由此可见，有两种葡萄酒尚未评价。

用这些工具，可以创建许多强大的组合，用于图的遍历与转换。假设我们想查找图中所有成对的朋友。为此，首先需要查找所有类型为 friends 的边，然后用 transform 操作，输出共享该边的两个人的名字。

```
g.V.outE('friends').transform{[it.outV.name.next(), it.inV.name.next()]}

==> [Patty, Tom]
==> [Patty, Alice]
```

在上述代码中，`transform` 闭包的返回值是包含两个元素的数组矢量（`[...]`）：`friends` 边的进出顶点。

为查找所有的人以及他们喜欢的酒，我们将 transform 之前步骤所输出的人（friends 边的顶点）转化为以两个元素对为其元素的列表：人的 name 与他喜欢的酒的列表。

```
g.V.both('friends').dedup.transform{
  [ it.name, it.out('likes').name.toList() ]
}

==> [Alice, [Prancing Wolf Ice Wine 2007]]
==> [Patty, []]
==> [Tom, [Prancing Wolf Ice Wine 2007, Prancing Wolf Kabinett 2002]]
```

习惯 Gremlin 无疑需要一点时间，特别是你以前从未用 Groovy 大量编程。一旦你找到窍门，就会发现 Gremin 是查询 Neo4j 的利器，表达力强且十分强大。

7.2.8　特定领域的步骤

图遍历固然很好，但是企业和组织倾向于用特定领域语言进行对话。比如，我们一般不会问“与葡萄酒顶点共享其向外边 `grape_type`，并将该边作为进入边的顶点是什么？”而会说“葡萄酒是什么品种的？”

其实，Gremlin 已经是特定于查询图数据库领域的一种语言，但是何不让它更为专门化呢？我们可以这么做，在 Gremlin 中创建新的步骤，这些步骤对于图中存储的数据，在语义上是有意义的。

让我们从新建名为 `varietal` 的步骤开始，它用来回答前面提出的问题。当在某个顶点上调用 `varietal()` 方法时，它会寻找类型为 `grape_type` 的向外的边，然后取出边那端的相关顶点。由于在这里会涉及 Groovy，因此我们先来看看创建步骤的代码，然后逐行进行描述。

```
neo4j/varietal.groovy
Gremlin.defineStep( 'varietal',
  [Vertex, Pipe],
  {_().out('grape_type').dedup}
)
```

第一行告诉 Gremlin 引擎，我们正要加入称为 varietal 的新步骤。第二行告诉 Gremlin，这个新建的步骤 varietal 会附加到 Vertex（顶点）和 Pipe（管道）类。最后一行则是神奇所在。事实上，它创建一个闭包，而新建的步骤就会执行该闭包中的代码。其中，下划线和圆括号表示当前的 pipeline 对象。从这个对象，我们通过 grape_type 边循到相关的邻节点——也就是，品种（varietal）节点。我们以 dedup 结尾，以移除任何可能重复的节点。

调用新建的步骤并无什么不同。比如，下面的代码会获取冰葡萄酒（ice wine）的品种名称。

```
g.V.filter{it.name=='Prancing Wolf Ice Wine 2007'}.varietal.name

==> riesling
```

我们再来尝试一次，编写一个常用的步骤：获取所有 friends 所喜爱的葡萄酒。

```
neo4j/friendsuggest.groovy
Gremlin.defineStep( 'friendsuggest',
  [Vertex, Pipe],
  {
    _().sideEffect{start = it}.both('friends').
    except([start]).out('likes').dedup
  }
)
```

和上次一样，给新步骤取名为 friendsuggest，并将它绑定到 Vertex（顶点）和 Pipe（管道）。这次，代码通过将当前的顶点或者 pipe 赋值给变量（start）——使用 sideEffect{start = it}函数，过滤出当前所处理的人。然后，再获取所有的 friends 节点，其中不包括当前所处理的人（显然，我们不想将 Alice 列为她自己的朋友）。

现在用 pipe 完成整个流程，调用新建步骤的方式与往常无异。

```
g.V.filter{it.name=='Patty'}.friendsuggest.name

==> Prancing Wolf Ice Wine 2007
==> Prancing Wolf Kabinett 2002
```

由于 varietal 和 friendsuggest 是普通的基于管道（Pipe-building）的步骤，因此可以将它们链接起来，成为更有趣的查询。下面的查询语句就是由两者组合而成，用于查找 Patty 的朋友最喜欢的葡萄酒品种：

```
g.V.filter{it.name=='Patty'}.friendsuggest.varietal.name

==> riesling
```

用 Groovy 元编程（metaprogramming）建立新的步骤是一种用于编写特定领域语言的强大力量。但是，正如 Gremin 本身，这需要一些实践来习惯与掌握。

7.2.9　更新、删除与完成

你已经插入一个图，然后遍历了它，但是，怎样更新和删除数据呢？只要你能找到想修改的顶点或者边，处理（更新或者删除）它们是很简单的。我们不妨为边 `likes` 增加属性 `weight`，以表示 Alice 有多喜欢 Prancing Wolf Ice Wine 2007。

```
gremlin> e=g.V.filter{it.name=='Alice'}.outE('likes').next()
gremlin> e.weight = 95
gremlin> e.save
```

同样，也能方便地删除数据。

```
gremlin> e.removeProperty('weight')
gremlin> e.save
```

在完成一天的学习和开始做作业之前，我们还要讨论如何清除数据库。

切记，完成今天的作业后，再运行下面的命令！

graph 对象（即 `g`）有用来删除顶点与边的函数，分别是函数 `removeVertex` 与 `removeEdge`。删除所有的顶点和边，图（数据库）也就析构了。

```
gremlin> g.V.each{ g.removeVertex(it) }
gremlin> g.E.each{ g.removeEdge(it) }
```

运行 `g.V` 与 `g.E`，可以验证是否所有的顶点与边都已删除。或者，也能运行十分危险的方法 `clear()` 来删除整个图。

```
gremlin> g.clear()
```

如果你正运行自己的 Gremlin 实例（Web 界面中的 Gremlin 实例除外），最好用方法 shutdown()明确地关闭图连接。

```
gremlin> g.shutdown()
```

不关闭图连接的后果是，数据库可能因此而崩溃。但是，通常来说，你只会在下次连接图的时候遇到麻烦。

7.2.10 第 1 天总结

今天，我们大概了解了图数据库 Neo4j——相比其他数据库，Neo4j 真够与众不同的。尽管我们没有讨论特定的设计模式，但是第一次开始用 Neo4j 进行工作的时候，我们的脑袋还是会嗡嗡作响，涌现各种灵感。只要你能在白板上画出来，你就能在图数据库中存储。

第 1 天作业

求索

1．将 Neo4j 的 wiki 加入书签。

2．将 Gremlin 的 wiki 或者 API 中的 Gremlin 步骤加入书签。

3．找到两个其他的 Neo4j shell（比如，admin 控制台中的 Cypher shell）。

实践

1．用其他 shell（比如，Cypher 查询语言）查询所有节点的 name。

2．删除你的图数据库中的所有节点与边。

3．创建一个代表你的家庭的新图。

7.3 第 2 天：REST、索引与算法

今天我们从 Neo4j 的 REST 接口开始，用 REST 创建节点与关系，然后建立索引并执行全文搜索。接着，我们会看一个插件，用于在服务器上通过 REST 执行 Gremlin 查询，于是，代码得以摆脱 Gremlin 控制台的束缚——甚至，索性在应用服务器与客户端中运行 Java 程序。

7.3.1 引入REST

正如 Riak、HBase、Mongo 与 CouchDB，Neo4j 发布的版本包含 REST 接口。所有这些数据库之所以支持 REST，原因之一是 REST 让我们以标准的连接接口，进行编程语言无关的交互。尽管 Neo4j 依赖于 Java，但是我们可以从一台独立的机器，完全不用管什么 Java，就能连接到 Neo4j 与之交互。而有了 Gremlin 插件，我们便能一睹基于 REST 的简洁查询语法之威力。

首先，你要对基本 URL 执行 `GET` 方法，获取根节点，由此可以检查 REST 服务器是

否正常运行。REST 运行的端口，与你昨天所用的 web admin 工具相同，路径是/db/data。我们会用向来不辱使命的 curl 程序来执行 REST 命令。

```
$ curl http://localhost:7474/db/data/
{
  "relationship_index" : "http://localhost:7474/db/data/index/relationship",
  "node" : "http://localhost:7474/db/data/node",
  "relationship_types" : "http://localhost:7474/db/data/relationship/types",
  "extensions_info" : "http://localhost:7474/db/data/ext",
  "node_index" : "http://localhost:7474/db/data/index/node",
  "extensions" : {
  }
}
```

这条命令会返回一个 JSON 对象，描述了其他命令的 URL，比如，节点操作或者索引。

7.3.2 用REST新建节点与关系

用 Neo4j 的 REST 接口建立节点和关系，与 CouchDB 或者 Riak 一样简单。新建节点只须对路径 db/data/node 以 POST 方法提交 JSON 数据。方便起见，每个节点都会有一个 name 属性，便于我们查看节点的信息：直呼其名即可。

```
$ curl -i -X POST http://localhost:7474/db/data/node \
-H "Content-Type: application/json" \
-d '{"name": "P.G. Wodehouse", "genre": "British Humour"}'
```

提交请求后，你会在 HTTP 头中取得节点路径，HTTP 体中则是关于节点的元数据（简洁起见，这里不做详述）。所有这些数据都可以检索，只须对给定的 HTTP 头属性 Location（或者元数据中的 self 属性）中的 URL 值调用 GET 方法。

```
HTTP/1.1 201 Created
Location: http://localhost:7474/db/data/node/9
Content-Type: application/json

{
  "outgoing_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/out",
  "data" : {
    "genre" : "British Humour",
    "name" : "P.G. Wodehouse"
  },
  "traverse" : "http://localhost:7474/db/data/node/9/traverse/{returnType}",
  "all_typed_relationships" :
```

```
    "http://localhost:7474/db/data/node/9/relationships/all/{-list|&|types}",
  "property" : "http://localhost:7474/db/data/node/9/properties/{key}",
  "self" : "http://localhost:7474/db/data/node/9",
  "properties" : "http://localhost:7474/db/data/node/9/properties",
  "outgoing_typed_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/out/{-list|&|types}",
  "incoming_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/in",
  "extensions" : {
  },
  "create_relationship" : "http://localhost:7474/db/data/node/9/relationships",
  "paged_traverse" :
    "http://localhost:7474/db/.../{returnType}{?pageSize,leaseTime}",
  "all_relationships" : "http://localhost:7474/db/data/node/9/relationships/all",
  "incoming_typed_relationships" :
    "http://localhost:7474/db/data/node/9/relationships/in/{-list|&|types}"
}
```

如果你想要节点的所有属性（不是元数据），可以在 GET 方法的节点 URL 后追加/properties，或者你想获取某个特定属性，可以在/properties 的基础上，再追加该属性的名字。

```
$ curl http://localhost:7474/db/data/node/9/properties/genre
"British Humour"
```

一个节点对我们来说是不够的，继续创建节点，其属性为["name" : "Jeeves Takes Charge", "style" : "short story"]。

由于 P.G. Wodehouse 写了短篇小说“Jeeves Takes Charge”，因此我们可以在已有的两个节点间建立关系。

```
$ curl -i -X POST http://localhost:7474/db/data/node/9/relationships \
-H "Content-Type: application/json" \
-d '{"to": "http://localhost:7474/db/data/node/10", "type": "WROTE",
  "data": {"published": "November 28, 1916"}}'
```

REST 接口的一大好处是，在前面 HTTP 体元数据的 create_relationship 属性中，已经展示了如何创建关系。在这种方式下，REST 接口往往是相互启发的。

7.3.3 查找路径

通过 REST 接口，以 POST 方式，向起始节点的/paths URL 提交数据，可以查找两个

节点之间的路径。POST 请求数据必须是一个 JSON 字符串，其中包含以下信息，你所查找路径的终止节点、你想循着的关系类型以及使用的路径查找算法。

比如，在这个例子中，从节点 9[1]沿着类型为`WROTE`的关系查找路径，并且使用最短路径（`shortestPath`）算法，在查找深度超过 10 的时候退出。

```
$ curl -X POST http://localhost:7474/db/data/node/9/paths \
-H "Content-Type: application/json" \
-d '{"to":"http://localhost:7474/db/data/node/10",
  "relationships": {"type" : "WROTE"},
  "algorithm":"shortestPath", "max_depth":10}'

[ {
  "start" : "http://localhost:7474/db/data/node/9",
  "nodes" : [
    "http://localhost:7474/db/data/node/9",
    "http://localhost:7474/db/data/node/10"
  ],
  "length" : 1,
  "relationships" : [ "http://localhost:7474/db/data/relationship/14" ],
  "end" : "http://localhost:7474/db/data/node/10"
} ]
```

其他可选的路径算法包括全路径(`allPaths`)算法、全简单路径(`allSimplePaths`)算法与迪科斯彻（`dijkstra`）算法。从在线文档[2]中可以找到这些算法的详细信息，但这些内容不在本书的讨论范围之内。

7.3.4 索引

与我们见过的其他数据库一样，Neo4j 能够通过建立索引，支持快速数据查找。但是，Neo4j 有其特殊之处。对于其他数据库的索引，所用的查询方式与没有索引时无异，但 Neo4j 不是这样，它的索引有不同的访问路径。原因在于，索引服务实际上是一个单独的服务。

最简单的索引是键-值（key-value）或者哈希（hash）形式的。可以用某些节点数据作为键，REST URL 作为值，该 URL 指向图中的节点。可以有任意数量的索引，把刚创建的这个索引命名为“authors”。URL 的末尾会包含我们想索引的 author 的名字，并传入节点 1 作为值（图形的 Wodehouse 节点，未必是 1）

[1] 原文是 node 1，看起来应该是 node 9。——译者注

[2] http://api.neo4j.org/current/org/neo4j/graphalgo/GraphAlgoFactory.html

```
$ curl -X POST http://localhost:7474/db/data/index/node/authors \
-H "Content-Type: application/json" \
-d '{ "uri" : "http://localhost:7474/db/data/node/9",
"key" : "name", "value" : "P.G.+Wodehouse"}'
```

检索节点只须简单地访问索引，不难发现，检索返回的不是我们设置的 URL，而是实际的节点数据。

```
$ curl http://localhost:7474/db/data/index/node/authors/name/P.G.+Wodehouse
```

除了键值索引，Neo4j 还提供了全文搜索倒排索引，所以可以这样执行查询：给我所有名字以“Jeeves”开头的图书。要建立这种索引，需要对整个数据集进行操作，而非之前那个单独的索引。与 Riak 一样，Neo4j 集成了 Lucene 来建立倒排索引。

```
$ curl -X POST http://localhost:7474/db/data/index/node \
-H "Content-Type: application/json" \
-d '{"name":"fulltext", "config":{"type":"fulltext","provider":"lucene"}}'
```

上面的 POST 请求会返回一个 JSON 响应，其中包含了新建索引的信息。

```
{
  "template" : "http://localhost:7474/db/data/index/node/fulltext/{key}/{value}",
  "provider" : "lucene",
  "type" : "fulltext"
}
```

现在，如果把 Wodehouse 加入全文索引，命令如下：

```
curl -X POST http://localhost:7474/db/data/index/node/fulltext \
-H "Content-Type: application/json" \
-d '{ "uri" : "http://localhost:7474/db/data/node/9",
"key" : "name", "value" : "P.G.+Wodehouse"}'
```

加入全文索引之后，搜索就如 Lucene 的索引 URL 查询语法一样简单。

```
$ curl http://localhost:7474/db/data/index/node/fulltext?query=name:P*
```

此外，也可以像上面的命令一样，对边建立索引；只需要用关系实例替代 URL 中的节点，比如，http://localhost:7474/db/data/index/relationship/published/date/1916-11-28。

7.3.5 REST与Gremlin

我们在第一天讨论了Gremlin，今天的前一半时间则花在REST接口上。如果你对究竟该用哪个感到疑惑，别担心，请往下看。Neo4j的REST接口有Gremlin插件（我们使用的Neo4j版本默认安装了该插件）[1]。可以通过REST发送在Gremlin控制台中可用的任何命令。因此，可以在Neo4j中使用这两个工具，并能按需做出选择。因为Gremlin适合复杂的查询，而REST则长于部署以及灵活性，所以两者相辅相成，是对很棒的组合。

以下代码会返回顶点名。只需将数据作为 JSON 字符串值发送到插件 URL，这里的查询语句置于 script 域中。

```
$ curl -X POST \
http://localhost:7474/db/data/ext/GremlinPlugin/graphdb/execute_script \
-H "content-type:application/json" \
-d '{"script":"g.V.name"}'

[ "P.G. Wodehouse", "Jeeves Takes Charge" ]
```

尽管从这里开始，代码示例都会使用 Gremlin，但请记住，可以选择使用 REST 代替。

7.3.6 大数据

到目前为止，我们处理的都是很小的数据集。现在，是时候看看 Neo4j 可以对大数据做些什么。

我们不妨从Freebase.com抓取数据集，探索一些电影数据。我们会使用由制表符作为分隔符的“performance”数据集合[2]。下载该文件，并使用下面的脚本——该脚本遍历每一行，在新节点或者已有节点之间创建关系（匹配是由索引中的名字决定的）。

注意，这个数据集包含大量电影信息，从卖座片到外国电影，还有成人娱乐电影。运行这个脚本需要安装 json 与 faraday Ruby gems。

```
neo4j/importer.rb
REST_URL = 'http://localhost:7474/'
HEADER = { 'Content-Type' => 'application/json' }

%w{rubygems json cgi faraday}.each{|r| require r}
```

[1] http://docs.neo4j.org/chunked/1.7/gremlin-plugin.htm
[2] http://download.freebase.com/datadumps/latest/browse/film/performance.tsv

```
# make a connection to the Neo4j REST server
conn = Faraday.new(:url => REST_URL) do |builder|
 builder.adapter :net_http
end

# method to get existing node from the index, or create one
def get_or_create_node(conn, index, value)
  # look for node in the index
  r = conn.get("/db/data/index/node/#{index}/name/#{CGI.escape(value)}")
  node = (JSON.parse(r.body).first || {})['self'] if r.status == 200
  unless node
    # no indexed node found, so create a new one
    r = conn.post("/db/data/node", JSON.unparse({"name" => value}), HEADER)
    node = (JSON.parse(r.body) || {})['self'] if [200, 201].include? r.status
    # add new node to an index
    node_data = "{\"uri\" : \"#{node}\", \"key\" : \"name\",
      \"value\" : \"#{CGI.escape(value)}\"}"
    conn.post("/db/data/index/node/#{index}", node_data, HEADER)
  end
  node
end

puts "begin processing..."

count = 0
File.open(ARGV[0]).each do |line|
  _, _, actor, movie = line.split("\t")
  next if actor.empty? || movie.empty?

  # build the actor and movie nodes
  actor_node = get_or_create_node(conn, 'actors', actor)
  movie_node = get_or_create_node(conn, 'movies', movie)

  # create relationship between actor and movie
  conn.post("#{actor_node}/relationships",
    JSON.unparse({ :to => movie_node, :type => 'ACTED_IN' }), HEADER)

  puts " #{count} relationships loaded" if (count += 1) % 100 == 0

end

puts "done!"
```

一些就绪，只要对下载的 `performance.tsv` 文件，运行这个脚本即可。

```
$ ruby importer.rb performance.tsv
```

处理完全部数据可能要花几个小时，但是可以在任何时候停下来，从而获得部分的电影/演员列表。如果用的是 Ruby 1.9，最好用 `builder.adapter:em_synchrony` 替换 `builder.adapter: net_http`，以便创建一个非阻塞的连接。

7.3.7　功能全面的算法

为了处理这个数据量很大[1]的电影数据集，我们暂时放下REST接口，重新使用Gremlin。

1. 舍我其谁，Kevin Bacon

我们来找点乐趣，实现一个更为著名的图算法：Kevin Bacon 算法。这个算法源于一个游戏——通过共同出演的电影，找出任意一个演员与 Kevin Bacon 的最短距离。举例来说，Alec Guinness 与 Theresa Russell 一起参演了《Kafka》，而 Theresa Russell 又在《Wild Things》中与 Kevin Bacon 合作。

在继续之前，打开 Gremlin 控制台，启动图。然后会用如下代码，创建 `costars` 自定义步骤。这与昨天的 `friendsuggest` 类似，下面的代码查找某个演员节点的联合主演（即，与一个演员所演出的电影相连的演员）。

```
neo4j/costars.groovy
Gremlin.defineStep( 'costars',
  [Vertex, Pipe],
  {
    _().sideEffect{start = it}.outE('ACTED_IN').
    inV.inE('ACTED_IN').outV.filter{
      !start.equals(it)
    }.dedup
  }
)
```

在 Neo4j 里，“查询”值不同于“遍历”图。其中蕴藏的好处是，一般来说，沿图遍历到的第一个节点距离起始节点最近（依据原始的边/节点距离，而非加权距离）。查询起始与终止节点，代码如下所示。

```
gremlin> bacon = g.V.filter{it.name=='Kevin Bacon'}.next()
gremlin> elvis = g.V.filter{it.name=='Elvis Presley'}.next()
```

[1] 在处理小规模问题时效率很低，这是因为算法时间效率中的常量很大，而问题往往规模很小。除非你知道你遇到的常常是复杂的情况，否则就让代码丑陋但是简单而高效吧。详细信息参见 http://users.ece.utexas.edu/～adnan/pike.html。——译者注

先查找某个演员的联合主演的联合主演的联合主演……经典的终止距离是六度，但实际上，在四度我们就能停下了（如果你没找到匹配的，可以重试）。这里我们可能循环图 4 次，查找所有“四度”的演员。我们会使用刚建立的 `costars` 步骤。

```
elvis.costars.loop(1){it.loops < 4}
```

只有能够到达 Bacon 的顶点才会保留。所有其他顶点都忽略。

```
elvis.costars.loop(1){
  it.loops < 4
}.filter{it.equals(bacon)}
```

为确保不回溯到 Kevin Bacon 节点做第二遍循环，需要剔出 bacon 节点的短路路径。或者，换句话说，只要循环不满 4 次且没有遇到 bacon 节点，就继续循环。然后输出到达每个 bacon 节点的路径。

```
elvis.costars.loop(1){
  it.loops < 4 & !it.object.equals(bacon)
}.filter{it.equals(bacon)}.paths
```

这样，只需取出可能路径列表中的第一条路径——最早到达的最短路径。>>记法用于弹出所有节点列表中的第一个元素。

```
(elvis.costars.loop(1){
  it.loops < 4 & !it.object.equals(bacon)
}.filter{it.equals(bacon)}.paths >> 1)
```

最后，得到每个顶点的名字，并用 Groovy 的 grep 命令，过滤掉任何值空的边数据。

```
(elvis.costars.loop(1){
  it.loops < 4 & !it.object.equals(bacon)
}.filter{it.equals(bacon)}.paths >> 1).name.grep{it}

==>Elvis Presley
==>Double Trouble
==>Roddy McDowall
==>The Big Picture
==>Kevin Bacon
```

我们不知道谁是 Roddy McDowall，但这正是图数据库的美妙之处。得到想要的答案，我们却不必了解每个细节。如果想要更为精彩的查询输出，尽你所能提炼你的 Groovy 代码吧，但是数据仍然在那里。

2. 随机遍历

在寻找大数据集中的范例时，一个有用的方法是“随机遍历”。可以从随机数产生器开始。

```
rand = new Random()
```

然后，可以过滤出占总数某个目标比例的对象。如果我们想只返回 Kevin Bacon 的大约三分之一电影，一共 60 部，取小于 0.33 的随机数即可。

```
bacon.outE.filter{rand.nextDouble() <= 0.33}.inV.name
```

最终返回的会是 Bacon 的作品中 20 个左右随机的电影名。

从 Kevin Bacon 做两度查询，即他的联合主演的联合主演，会创建一个相当大的列表（该数据集里有不止 300 000 条）。

```
bacon.outE.inV.inE.outV.loop(4){
it.loops < 3
}.count()

==> 316198
```

但是如果你只需要该列表的 1/100，增加一个过滤器。同时，需要注意，过滤器本身也是一个步骤，所以要将 loop 数字加一。

```
bacon.outE{
  rand.nextDouble() <= 0.01
}.inV.inE.outV.loop(5){
  it.loops < 3
}.name
```

我们会在结果中找到 Elijah Wood，根据 Bason 路径算法，以两步找到他是合理的（Elijah Wood 在《Deep Impact》中与 Ron Eldard 合作，Ron 与 Kevin Bacon 一起参演了《Sleepers》）。

3. 关于中心度

中心度是对全图中单个节点的度量。比如，如果我们想基于每个节点到其他所有节点的距离，度量其重要性，这就需要一个中心度算法。

最著名的中心度算法莫过于 Google 的 PageRank，但这也分为若干类型。我们将执行一个称为特征向量中心度（eigenvector centrality）的简单版本，它仅计算某节点的进出边的数目。我们会给每个演员节点一个数字，表示其出演的角色的数量。

我们需要用 groupCount()产生一个 map，还需要一个变量 count 表示循环的最大次数。

```
role_count = [:]; count = 0
g.V.in.groupCount(role_count).loop(2){ count++ < 1000 }; ''
```

role_count 映射的键为顶点，值为该顶点拥有的边的数目。查看这个 map 最简便的方式就是将它排序。

```
role_count.sort{a,b -> a.value <=> b.value}
```

在排序之后，排列在最后的是出演最多作品的演员。在该数据集中，这项荣誉属于传奇配音演员 Mel Blanc，以 424 部代表作问鼎（可以用这条命令把这些影片都列出来：g.V.filter{it.name=='Mel Blanc'}.out.name）。

外部算法

你可以自己写算法，但大部分算法都是现成的。JUNG（Java Universal Network/Graph）框架包含了常用的图算法，以及图建模与可视化的若干工具。我们得感谢 Gremlin/Blueprint 项目，有了它我们得以方便地访问 JUNG 的算法，如，PageRank、HITS、Voltage、centrality algorithms，以及 graph-as-a-matrix 工具。

要使用JUNG框架，需要将Neo4j图包装成JUNG图[1]。而访问JUNG图，有两种选择：下载Blueprint与JUNG的全部jar文件，并安装到Neo4j服务器的lib目录，然后重启服务器；或者下载预先打包好的Gremlin控制台。在此，我们推荐后者，因为这会避免不少寻找若干Java压缩文件（jar）的麻烦。

假设你已经下载了 gremlin 控制台，关闭 neo4j 服务器，启动 Gremlin。你必须创建一个 Neo4jGraph 对象，并将它指向你的 data/graph 目录。

```
g = new Neo4jGraph('/users/x/neo4j-enterprise-1.7/data/graph.db')
```

我们会保留 Gremlin 图的名字不变，依然为 g。而该 Neo4jGraph 对象需要包裹于 GraphJung 对象 j 之中。

```
j = new GraphJung( g )
```

Kevin Bacon 之所以被选作终极路径的终点，部分原因是他与其他演员关联较多——他与当红影星一起出演电影。但更为重要的是，他其实不必亲自参演很多角色，与其他演员

[1] http://blueprints.tinkerpop.com

产生联系，而只要简单地与本身就有很多联系的演员相关联即可。

这就产生了一个问题：我们能否找到比 Kevin Bacon 更为适合的演员，来度量其与其他演员的距离？

JUNG 包含名为 BarycenterScorer 的评分算法，该算法基于每个顶点到其他所有顶点的距离，为该顶点评分。如果 Kevin Bason 的确是最佳选择，我们预期他的分数是最低的，这意味着他与所有其他演员"最近"。

因为 JUNG 算法只应用于演员，所以构建一个转换器将所有的演员节点滤出。EdgeLabelTransformer 会滤出带有指向该算法的边 ACTED_IN 的节点。

```
t = new EdgeLabelTransformer(['ACTED_IN'] as Set, false)
```

下一步，需要导入算法本身，传入 GraphJung 与转换器。

```
import edu.uci.ics.jung.algorithms.scoring.BarycenterScorer
barycenter = new BarycenterScorer<Vertex,Edge>( j, t )
```

之后，就能获得任意节点的 BarycenterScorer 评分。我们来看看 Kevin Bacon 的分数是多少。

```
bacon = g.V.filter{it.name=='Kevin Bacon'}.next()
bacon_score = barycenter.getVertexScore(bacon)

~0.0166
```

一旦有了 Kevin Bacon 的分数，就可以遍历所有顶点并将低于这个分数的顶点保存下来。

```
connected = [:]
```

为数据库中的每个演员计算 BarycenterScorer 评分会花费相当长的时间。所以，简化一下，对 Kevin 的联合主演执行该算法。这会执行若干分钟。具体情况取决于硬件条件。尽管 BarycenterScorer 算法相当快，但 Bacon 的每个联合主演的计算时间累计下来，还是很可观的。

```
bacon.costars.each{
  score = barycenter.getVertexScore(it);
  if(score < bacon_score) {
    connected[it] = score;
  }
}
```

在映射 connected 中的所有键都表示优于 Kevin Bacon 的选择。但是，最好有个我们熟识的名字，所以我们不妨把他们都输出，挑选个我们喜欢的。你得到的输出可能与我们的不同，因为开放的电影数据集总是在变的。

```
connected.collect{k,v -> k.name + " => " + v}

==> Donald Sutherland => 0.00925
==> Clint Eastwood => 0.01488
...
```

Donald Sutherland 作为一名可敬的演员，分数约为 0.009 25。因此，假设 Donald Sutherland 的六度比 Kevin Bacon 的传统六度更适合与你的朋友玩这个游戏。

有了 j 图，现在可以执行任何 JUNG 算法，比如，PageRank。像 BarycenterScorer 一样，需要先导入类。

```
import edu.uci.ics.jung.algorithms.scoring.PageRank
pr = new PageRank<Vertex,Edge>( j, t, 0.25d )
```

JUNG 算法的完整列表可以在其在线 Javadoc API 中找到。其中，逐渐加入越来越多的算法，因此在自己实现算法之前，不妨先看看有没有现成的。

7.3.8　第 2 天总结

在第 2 天里，我们学习了 REST 接口，从而扩宽了与 Neo4j 交互的能力。我们还讨论了如何使用 Gremlin 插件，于是我们可以在服务器上执行 Gremlin 代码并通过 REST 接口返回结果。接着，我们处理了更大的数据集，最后通过几个算法深入挖掘了这些数据。

第 2 天作业

查找

1. 将 Neo4j 的 REST API 文档加入收藏夹。

2. 将 JUNG 项目的 API 与它实现的算法加入收藏夹。

3. 为你最喜爱的编程语言找到一个绑定或者 REST 接口。

完成

1. 将 Kevin Bacon 算法的路径发现部分，转化为一个步骤。然后，实现一个 Groovy 函数，输入参数为图以及两个名字，功能是比较距离。

2．对一个节点（或者 API 所需的数据集），选择并运行一个 JUNG 算法。

3．安装你选择的驱动程序，用它管理你的公司图——图中包含公司人员以及他们的角色，边则描述了人员间的交互关系（reports to，works with）。如果你的公司很大，就用你所在的团队代替；如果你的公司太小，就引入一些客户。然后，通过计算到其他所有节点的最短距离，找到整个组织架构中关联最紧密的人。

7.4　第 3 天：分布式高可用性

我们将学习如何使 Neo4j 更适用于关键任务，并以此总结对于 Neo4j 的研究。首先，我们会看到 Neo4j 如何通过遵守 ACID 的事务，来保证数据的稳定。然后，我们会安装并配置 Neo4j 高可用性（High Availability，HA）集群，提升高强度读操作负载时的可用性。最后，我们将学习备份策略，保证数据的安全。

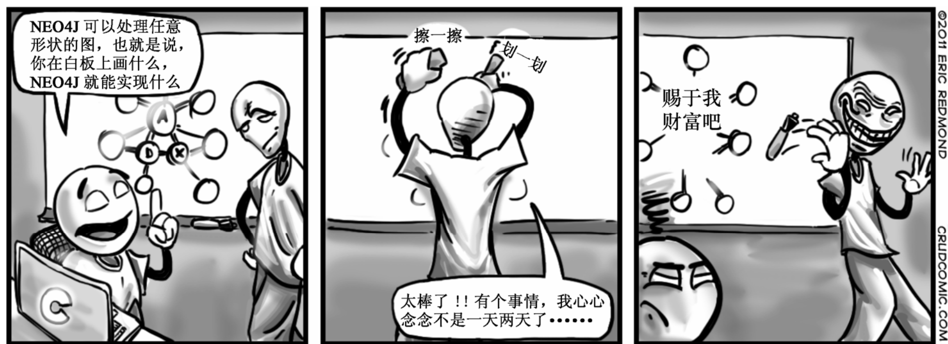


图 7-9　白板友好

7.4.1　事务

Neo4j 是一个支持原子性、一致性、隔离性以及持久性（ACID）事务的数据库，这与 PostgreSQL 是类似的。这使 Neo4j 成为重要数据存储的一种不错选择，而对于这一领域我们往往会采用关系数据库。与我们所认识的事务一样，Neo4j 的事务也是“全部或全无”（all-or-nothing）的操作。一旦某个事务开始执行，每个后续操作都会作为一个原子单位，成功或者失败——一处失败即是全部失败。

关于事务如何处理的细节涉及底层中名为 Blueprint 且基于 Neo4j 的包装项目，这超出了 Gremlin 的范围。而具体细节随版本变化有所不同。这里用的是 Gremlin 1.3，对应 Blueprint 1.0。如果你用了不同的版本，可以在 Blueprint API Javadocs 里找到对应的具体细节。

与 PostgreSQL 一样，单行的基本函数自动成为隐式事务。为演示多行事务，需要标记图对象以关闭自动事务模式，让 Neo4j 知道我们要手动处理事务。可以通过函数 setTransactionMode()改变事务模式。

```
gremlin> g.setTransactionMode(TransactionalGraph.Mode.MANUAL)
```

利用 startTransaction()和 stopTransaction(conclusion)，可以启动停止图对象的事务。在停止事务时，还需要标记事务是否成功执行。否则，Neo4j 可能回滚从事务开始起的所有命令。把事务放在 try/catch 代码块中是个好主意，这能确保一旦出现异常，就会触发回滚。

```
g.startTransaction()
try {
  // execute some multi-step graph stuff here...
  g.stopTransaction(TransactionalGraph.Conclusion.SUCCESS)
} catch(e) {
  g.stopTransaction(TransactionalGraph.Conclusion.FAILURE)
}
```

如果你想在 Gremlin 之外直接操作并使用 Neo4j 的 EmbeddedGraphDatabase，你就可以用事务的 Java API 语法。而一旦用 Java 或者底层实现为 Java 的语言（像 JRuby）写代码，就不得不用相应的代码风格。

```
r = g.getRawGraph()
tx = r.beginTx()
try {
  // execute some multistep graph stuff here...
  tx.success()
} finally {
  tx.finish()
}
```

上面提到的两种方式都能提供 ACID 事务的全面保证。即使系统发生故障，也能保证任何写操作在服务器恢复时全部回滚。如果不需要手动处理事务，最好将事务模式保持为 TransactionalGraph.Mode.AUTOMATIC。

7.4.2　高可用性

“能实现大规模图数据库吗？”可以，但是有若干前提，高可用性模式就是 Neo4j 支持大规模图数据库的一种方式。因为对一个从节点的一次写入操作不会立刻同步到其他所有从节点，所以存在这样的风险——在一段时间内（最终还是能保证一致性的），会失去数据

一致性（CAP 意义上的一致性）。HA 会导致事务无法完全遵循 ACID。因此，Neo4j 的 HA 只是作为增加读操作能力的一种解决方案。

正如 Mongo 一样，集群中的服务器会选出一个主节点，作为数据的主副本。然而，与 Mongo 不同的是，从节点接受写操作。对从节点的写操作会与主节点同步，然后进一步扩散到其他从节点。

7.4.3 HA集群

为了应用 Neo4j 的 HA，必须先建立一个集群。Neo4j 使用名为 Zookeeper 的外部集群协调服务，它是从 Apache Hadoop 项目衍生而来的另一个出色项目，是一个协调分布式应用的通用服务。它被 Neo4j HA 用于生命周期活动的管理。每个 Neo4j 服务器都有自己的任务协调者，管理其在集群中的职责——如图 7-10 所示。

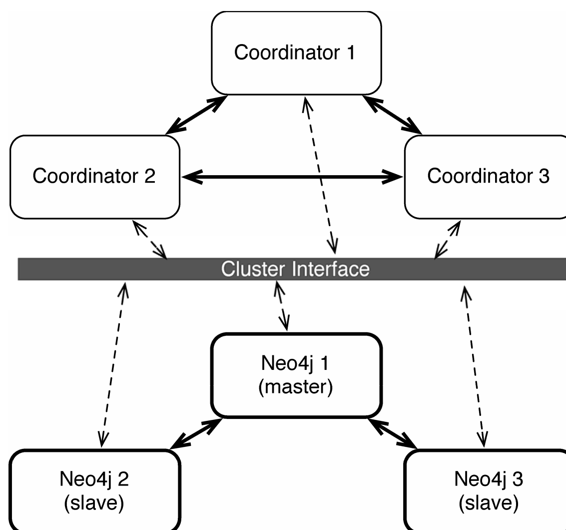


图 7-10 三个服务器组成的 Neo4j 集群及其协调者

幸运的是，Neo4j企业版捆绑了Zookeeper以及一些帮我们配置集群的文件。我们打算运行 3 个Neo4j企业版 1.7 的实例。你可以从网站下载适合于所用操作系统的副本（确保选择正确的版本）[1]，然后解压，并创建另两个副本。我们以 1、2 与 3 作为实例的后缀，并以此引用它们。

```
tar fx neo4j-enterprise-1.7-unix.tar
```

[1] http://neo4j.org/download/

```
mv neo4j-enterprise-1.7 neo4j-enterprise-1.7-1
cp -R neo4j-enterprise-1.7-1 neo4j-enterprise-1.7-2
cp -R neo4j-enterprise-1.7-1 neo4j-enterprise-1.7-3
```

现在，我们有同一数据库的三个相同的副本。

一般来说，需要在每个服务器上放置一个副本，并配置集群，让每个服务器都知道其他服务器的存在。但是，由于我们在本地运行这些数据库副本，因此只要在不同的目录与不同的端口运行它们就可以了。

按照下面的 5 步创建集群，首先配置 Zookeeper 集群协调者，然后配置 Neo4j 服务器。

1）为每个协调者服务器设置唯一的 ID

2）配置每个协调者服务器，使它与其他协调者服务器以及它所属的 Neo4j 服务器交互。

3）启动三个协调者服务器。

4）配置每个 Neo4j 服务器以 HA 模式运行，分配唯一的端口，并让它们知道集群的存在。

5）启动三个 Neo4j 服务器。

Zookeeper 通过每个服务器在集群中的唯一 ID 记录其状态。ID 的数字是存于文件 `data/coordinator/myid` 中的唯一值。对服务器 1，用默认值 1；服务器 2 和服务器 3 分别设置为 2 与 3。

```
echo "2" > neo4j-enterprise-1.7-2/data/coordinator/myid
echo "3" > neo4j-enterprise-1.7-3/data/coordinator/myid
```

我们还必须简单描述集群内部的一些通信设置。每个服务器都会有一个名为 `conf/coord.cfg` 的文件。我们会看到变量 `server.1` 的默认值为代表服务器的 `localhost` 与两个端口：quorum 选举端口（`2888`）与主节点选举端口（`3888`）。

1. 构建集群

Zookeeper quorum 是集群中的一组服务器，以及它们之间相互通信的端口（不要与 Riak 中的 quorum 混淆，Riak 中的 quorum 意指保证一致性的数量）。主节点选举端口在主服务器宕机时使用——这个特殊端口用来让余下的正常服务器选出一个新的主服务器。

保留变量 `server.1` 的默认值，并将后续端口赋给 `server.2` 与 `server.3`。服务

器 1、2 和 3 中的文件 coord.cfg 必须包含相同的三行。

```
server.1=localhost:2888:3888
server.2=localhost:2889:3889
server.3=localhost:2890:3890
```

最后，还必须设置 Neo4j 可能连接的公共端口。因为这个 `clientPort` 端口默认为 2181，所以保留服务器 1 的值。对服务器 2 和服务器 3，分别设置为 `clientPort=2182` 以及 clientPort=2183。如果你的机器已经使用了这些端口中的某个，可以随时按需更改，但我们依然会用被占用的端口描述后面的步骤。

2. 协调

用 Neo4j 团队提供的便捷脚本启动 Zookeeper 协调者。然后在每台服务器的目录中运行如下命令：

```
bin/neo4j-coordinator start
Starting Neo4j Coordinator...WARNING: not changing user
  process [36542]... waiting for coordinator to be ready. OK.
```

现在，协调者服务运行了，但是 Neo4j 并没有启动。

3. 连线 Neo4j

下一步，需要设置 Neo4j，让它运行于 HA 模式，然后再连接到协调者服务器。打开文件 `conf/neo4j-server.properties`，对于每个 Neo4j 实例都增加如下一行：

```
org.neo4j.server.database.mode=HA
```

这行配置使 Neo4j 运行于 HA 模式；截止目前，我们一直还运行在 SINGLE 模式。当编辑这个配置文件时，把 Web 服务器端口设置成唯一的数字。一般来说，就用默认端口 7474，但是由于在同一个机器上运行三个 Neo4j 实例，因此不能让 http/https 端口重复使用。对于服务器 1 用端口 7471/7481，对于服务器 2 用端口 7472/7482，对于服务器 3 用端口 7473/7483。

```
org.neo4j.server.webserver.port=7471
org.neo4j.server.webserver.https.port=7481
```

最后，让每个 Neo4j 实例连接到其中一个协调者服务器。如果打开了服务器 1 的配置文件 `conf/neo4j.properties`，就会看到若干行以 `ha` 开头的注释的内容。这些其实是关于 HA 的设置，描述了 3 件事：当前机器的号码（ID）、zookeeper 服务器列表以及 neo4j 服务器用来与其他服务器通信的端口。

对于服务器 1，为 neo4j.properties 属性添加如下字段：

```
ha.server_id=1
ha.coordinators=localhost:2181,localhost:2182,localhost:2183
ha.server=localhost:6001
ha.pull_interval=1
```

对于另两个实例，配置是类似的，只需注意两点：对于服务器 2ha.server_id=2，对于服务器 3 则 ha.server_id=3；另外，ha.server 必须使用不同的端口（服务器 2 用 6002，服务器 3 用 6003）。再次说明，如果 Neo4j 实例运行于不同的机器，端口是不必修改的。服务器 2 的配置包含如下内容（对于服务器 3 类似）：

```
ha.server_id=2
ha.coordinators=localhost:2181,localhost:2182,localhost:2183
ha.server=localhost:6002
ha.pull_interval=1
```

把 pull_interval 设置为 1，意为每个从节点检查主节点更新的间隙为 1 秒。一般来说，不会把值设置得这么小，之所以这么设置，是为了看到插入的演示数据会立刻更新到各个节点。

Neo4j HA 服务器配置完毕后，就该启动了。与协调者服务器的启动脚本一样，在安装目录中启动每个 neo4j 实例。

```
bin/neo4j start
```

通过查看日志文件的后若干行，你可以看到服务器的输出。

```
tail -f data/log/console.log
```

每个实例都会连接到它相应的协调者服务器。

4. 验证集群状态

首先启动的服务器会成为主服务器——可能是服务器 1，也可能不是。可以打开相应 Neo4j实例的Web管理页面，验证其是否为主服务器（我们之前将管理端口设置为 7471）。点击页面顶端的Server Info链接，以及侧边菜单的High Available[1]。

High Available 下的属性列出了这个集群的信息。如果这个服务器是主服务器，对应的属性为真；如果不是主服务器，可以在 InstancesInCluster 下找到哪个服务器成为主服

[1] http://localhost:7471/webadmin/#/info/org.neo4j/High%20Availability/

务器。

列出的内容包括每个相互连接的服务器，服务器的 ID（machine ID），服务器是否为主服务器以及一些其他信息。

5. 验证副本

集群启动并开始运行，你可以验证服务器是否正确地维护副本。如果一切正常，任何对从服务器的写操作都会传达主服务器节点，并且最终应用于其他的从服务器。打开三个服务器的 Web 控制台，可以使用内置的 Gremlin 控制台。注意，Gremlin 中的图对象已经变了，其中包含一个 HighlyAvailableGraphDatabase。

```
g = neo4jgraph[HighlyAvailableGraphDatabase [/…/neo4j-ent-1.7-2/data/graph.db]]
```

为测试服务器，我们打算为将几个节点加入到新图里，其中包含几个著名悖论的名字。在某个从服务器的控制台中，把根节点设置为芝诺悖论（Zeno's paradox）。

```
gremlin> root = g.v(0)
gremlin> root.paradox = "Zeno's"
gremlin> root.save
```

然后，转向主服务器的控制台，输出顶点 paradox 值。

```
gremlin> g.V.paradox
==> Zeno's
```

这时，如果你切换到其他从服务器，增加罗素悖论（Russell's paradox），然后查看 paradox，会看到在这第二个从服务器上显示了两个节点。而事实上，只在这个服务器中，直接添加过一个节点。

```
gremlin> g.addVertex(["paradox" : "Russell's"])
gremlin> g.V.paradox
==> Zeno's
==> Russell's
```

如果数据的变化没有传达某个从服务器，你可以回到 Server Info、High Availability 页面。查看 `lastCommittedTransactionId` 中的所有实例。如果其中的数值是相等的，则表明系统数据保持一致性。数值越小，该服务器上的数据版本越老。

6. 主服务器选择

一旦关闭主服务器，并在剩下的某个服务器上刷新服务器信息，你会看到另一个服务

器被选为新的主服务器。再次启动之前关闭的主服务器，会将该服务器重新加入集群，只是不再作为主服务器（直到目前的主服务器关闭）。

HA 使得读操作负载很重的系统能够在多个服务器之间处理图副本，从而分摊了负载。尽管集群作为一个整体只会在最终达到数据一致性，但是有些诀窍有助于减少读到过期数据的几率，例如，给服务器分配会话（session）。有了好的工具、规划以及设置，可以构建足够大的图数据库，处理十亿数量级的顶点与边，以及几乎任意数量的请求。在此基础上，只需增加定期的备份，你就得到了建立一个可靠生产系统的秘诀。

7.4.4 备份

备份是任何专业数据库应用不可缺少的方面。虽然，在使用副本的同时，亦有效引入了备份，但是夜间运行的场外（off-site）备份永远是数据灾难恢复的良药。要知道，为机房火灾或者将整栋大楼震为废墟的地震准备预案，是极其困难的。

Neo4j 企业版提供了名为 neo4j-backup 的简单备份工具。

对于 HA 服务器，最强大的备份方法莫过于编写一个全备份的命令，将数据库文件从集群复制到挂载的磁盘，成为一个带有日期戳的文件。将副本指向集群中的每个服务器，可以保证你获取最新的可用数据。创建的备份目录实则是一个完全可用的副本。如果你需要恢复数据，只需用备份目录替换每个服务器的数据目录，就这么简单。

第一次你必须做一个完整的备份。这里把 HA 集群备份到以今天的日期结尾的目录（使用*nix 操作系统的 date 命令）。

```
bin/neo4j-backup -full -from ha://localhost:2181,localhost:2182,localhost:2183 \
-to /mnt/backups/neo4j-`date +%Y.%m.%d`.db
```

如果你没有运行于 HA 模式，只要把 URI 中的模式改为 single。一旦做好全备份，就可以选择进行增量备份，仅存储上次备份以来变化的部分。如果我们想在半夜对单个服务器进行全备份，之后每两小时抓取增量备份，可以执行如下命令：

```
bin/neo4j-backup -incremental -from single://localhost \
-to /mnt/backups/neo4j-`date +%Y.%m.%d`.db
```

但是，务必牢记，增量备份只有在全备份目录基础之上才起作用，所以确保之前的全备份命令在同一天运行过。

7.4.5 第 3 天总结

今天我们讨论了保持 Neo4j 数据稳定的一些方法，包括遵循 ACID 的事务处理、高可用性以及备份工具。

必须注意到的是，我们今天使用的全部工具都需要 Neo4j 的企业版，并且使用了两个授权——GPL 与 AGPL。如果希望你的服务器是闭源的，就得研究转向社区版，或者从 Neo Technology（Neo4j 所属的公司）获取 OEM。联系 Neo4j 团队以得到更多信息。

第 3 天作业

查找

1．找到 Neo4j 的授权说明

2．回答问题，“支持的最大节点数是多少？”（提示：这在网站文档的 Questions & Answers 中。）

完成

1．在三个物理服务器上的 Neo4j 实例之间，相互复制。

2．用 Apache 或者 Nginx 之类的 Web 服务器建立负载均衡器，并用 REST 接口连接到集群。执行 Gremlin 脚本命令。

7.5 总结

Neo4j 是图数据库分类（相对较少的类别）中最好的开源实现。图数据库关注数据之间的关系，而不是值之间的共同特征。对图数据建模很简单。你只要创建节点和它们之间的关系，并可选地挂上键-值对。查询很简单，只要声明如何从开始节点遍历图。

7.5.1 Neo4j的优点

Neo4j 开源图形数据库最好的例子之一。图形数据库对于非结构化数据是完美的，在某些方面甚至好过基于文档数据存储（面向文档的数据库）。Neo4j 不但没有类型与模式的

概念，而且它对于数据关联的方式没有限制。从最好的意义上来说，Neo4j 是完全自由的。目前，Neo4j 支持 344 亿个节点以及 344 亿个关系，足够应对任何领域的应用（Facebook 有 8 亿用户，Neo4j 能够在单个图中为其每个用户持有 42 个节点）。

分布式 Neo4j 提供了若干种工具，可以整合 Lucene 进行快速查找，也可以用像 Gremlin 以及 REST 接口这样易用（相对于一些晦涩的）的语言扩展。除了易用性，Neo4j 也很快速。不像关系数据库中的正交操作或者其他数据库中的 map-reduce（映射-规约）操作，图遍历是常量时间复杂度的。在 Neo4j 中，数据仅仅是一步之遥的节点，而对于我们操作过的其他数据库，往往需要做正交操作，然后过滤期望的结果。不用在意图变得多大；从节点 A 到节点 B 永远只需要一步，只要它们是相互关联的（即存在 relationship）。最后，企业版通过 Neo4j 的 HA 功能，提供了高可用性以及应对密集读操作的能力。

7.5.2　Neo4j的缺点

Neo4j 确实有一些缺点。Neo4j 中的边无法从一个顶点回指其本身。我们也发现它所选择的术语增加了交流时的复杂性（节点而不是顶点，关系而不是边）。尽管 HA 在创建副本时表现出色，但是它只能将完整的图复制到其他服务器。目前来说，Neo4j 不能划分子图，这还是会限制图的大小（不过，公平地说，目前的限制也是以百亿计的）。最后，如果你在找商业友好的开源授权（比如 MIT），Neo4j 可能不适合你。而社区版（我们在前两天用的版本）是 GPL 的，如果你想在实际生产环境中使用企业版工具（包括 HA 和备份），你可能不得不购买授权了。

7.5.3　Neo4j之于CAP

如果你选择分布式系统，那么名为“高可用性”的集群无疑会透露其策略。Neo4j 的 HA 是关于可用性与分区容忍性（AP）的。每个从节点只会返回它目前所包含的数据，而这些数据可能暂时与主节点失去同步。尽管，通过增加从节点的同步频率，你只能减小更新的延迟，但是从技术上说，最终还是会达到数据的一致。这也是 Neo4j 的 HA 适用于偏重读操作需求的原因。

7.5.4　结束语

如果你还未习惯图数据的建模，那么 Neo4j 的简单反而让人懊恼。Neo4j 提供了

强大的开源 API 并经过多年生产应用的检验，但是用户数依然较少。由于图形数据库与人类概念化数据的方式有着自然的联系，因此我们把目前的状况归咎于人们还缺少相应的知识。我们会把家庭想象成树，把我们的朋友们想象为图；可我们大多数人不会把个人关系想象成自引用的数据类型。对某类问题，如社交网络，Neo4j 是显而易见的选择。此外，你也应认真思考不那么明显的问题——或许，你会惊喜于其强大与易用之处。

第 8 章

Redis

Redis 就像润滑油。润滑油通常用于润滑系统的各个运转部件，通过减少摩擦，保持它们运行顺畅，并加快其整体功能。不论哪种系统构造，加点润滑油很可能有改善。有时候，只要明智地使用 Redis，就能搞定你的问题。

Redis（REmote DIctionary Service，远程字典服务）最早发布于 2009 年，是一个简单易用的键值对存储库，带有一套成熟的命令。若论速度，很多数据库难出其右。它读取速度快，写入速度更快，根据某些基准测试，每秒可处理高达 10 万次 `SET` 操作。Redis 的创始人是萨尔瓦托·勒圣菲利波（Salvatore Sanfilippo），他把该项目称为“数据结构服务器”，以反映其对复杂数据类型及其他功能的细致处理。这不只是一个超快的键值对存储系统，学习它，将使我们对现代数据库的了解更加完整。

8.1 数据结构服务器存储库

将 Redis 精确归类可能有点困难。当然，从基本层面上说，它是一个键-值对存储库。但这种简单的说法并不全面。虽然 Redis 没有达到文档型数据库的程度，但它支持高级的数据结构。它支持基于集合的查询操作，但不支持关系数据库中同样的粒度或类型。当然，它很快，为了速度而在持久性方面作出了让步。

Redis 是高级数据结构服务器，此外，它也是阻塞队列（或栈）和发布-订阅系统。它支持可配置的到期策略、持久性级别，以及复制选项。所有这些使得 Redis 不仅是某类数据库中的一员，更是有用的数据结构算法和程序的工具包。

Redis 有丰富的客户端库，在许多编程语言中可以很方便地使用它。它不仅易用，还是一种乐趣。如果说 API 是程序员的用户体验，那么在现代艺术博物馆中 Redis 应该和 Mac

Cube 放在一起。

在第 1 天和第 2 天，我们将探讨 Redis 的功能、约定和配置。与往常一样，从简单的 CRUD 操作开始，我们会很快转向更高级的操作，涉及更强大的数据结构：列表、哈希表、集合和有序集合。我们将创建事务，并操作数据有效期的特征。我们会用 Redis 创建一个简单的消息队列，并探讨其发布-订阅功能。然后，我们将深入探讨 Redis 的配置和复制选项，学习如何在数据持久性和速度之间，取得适合应用程序的平衡。

数据库常常彼此配合使用，这种趋势在增加。把 Redis 安排在本书的最后介绍，这样我们就可以用这种方式使用它。在第 3 天，我们将构建我们的终极系统，一个功能丰富的多数据库的音乐解决方案，包括 Redis、CouchDB、Neo4j 和 Postgres，用 Node.js 将它们结合在一起。

8.2　第 1 天：CRUD 与数据类型

对 Redis 的开发团队来说，命令行界面（Comnand-Line Interface，CLI）最重要（也深受世界各地用户的喜爱），所以我们将在第 1 天讨论 Redis 的许多命令，Redis 中可用的命令共用 124 条。最重要的是理解它复杂的数据类型，以及除了简单的“检索此键的值”之外，如何通过更多的方式查询。

8.2.1　入门指南

Redis可以通过一些包管理工具来安装，如Mac的Homebrew，但是编译生成也不难。[1] 我们将使用 2.4 版。安装之后，可以通过调用下面的命令启动服务器：

```
$ redis-server
```

在默认情况下，它不会在后台运行，但是可以在命令后加上 &，实现后台运行，或者也可以打开另一个终端。接下来运行命令行工具，它应该自动连接到默认的 6379 端口。连接后，尝试来 ping 服务器。

```
$ redis-cli
redis 127.0.0.1:6379> PING
PONG
```

如果无法连接，你就会收到一条错误消息。输入 help 命令将显示帮助选项列表。输入

[1] http://redis.io

help 后面跟一个空格，然后输入任何命令，会给出该命令的帮助。如果你不知道任何一条 Redis 命令，按 Tab 键会循环列出所有的选项。

```
redis 127.0.0.1:6379> help
Type: "help @<group>" to get a list of commands in <group>
      "help <command>" for help on <command>
      "help <tab>" to get a list of possible help topics
      "quit" to exit
```

今天，我们要用 Redis 构建一个短 URL 服务的后端，类似于 tinyurl.com 或 bit.ly。短 URL 服务将很长的 URL 映射到自己域名中的短 URL，例如，将 `http://www.myveryververylongdomain. com/somelongpath.php` 映射到 `http://bit.ly/VLD`。当用户访问这个短 URL 时，会重定向到映射前的长 URL，这样用户就不需要发送长字符串，同时也为短 URL 的创造者提供了一些统计数据，例如，访问次数。

在 Redis 中，可以使用 `SET` 命令，将短码 `7wks` 作为键，将 `http://www.sevenweeks.org` 作为值。SET 总是需要两个参数，一个键和一个值。检索值，只需要 `GET` 命令和键名。

```
redis 127.0.0.1:6379> SET 7wks http://www.sevenweeks.org/
OK
redis 127.0.0.1:6379> GET 7wks
"http://www.sevenweeks.org/"
```

为了减少通信开销，也可以使用 `MSET` 设置多个值，比如，任何数量的键－值对。这里将 Google.com 映射到 `gog`，Yahoo.com 映射到 `yah`。

```
redis 127.0.0.1:6379> MSET gog http://www.google.com yah http://www.yahoo.com
OK
```

相应地，`MGET` 使用多个键，返回值是一个有序列表。

```
redis 127.0.0.1:6379> MGET gog yah
1) "http://www.google.com/"
2) "http://www.yahoo.com/"
```

虽然 Redis 存储字符串，但它也能识别整数，并提供一些简单的整数操作。如果我们想记录数据集中短键不断增长的总数，可以创建一个 count 变量，随后使用 `INCR` 命令递增它。

```
redis 127.0.0.1:6379> SET count 2
```

```
OK
redis 127.0.0.1:6379> INCR count
(integer) 3
redis 127.0.0.1:6379> GET count
"3"
```

虽然 GET 以字符串的形式返回 count，但 INCR 将它识别为一个整数，并对其加一。如果试图对任何非整数递增，结果就不妙了。

```
redis 127.0.0.1:6379> SET bad_count "a"
OK
redis 127.0.0.1:6379> INCR bad_count
(error) ERR value is not an integer or out of range
```

如果该值不能解析为整数，Redis 将正确地指出问题。还可以用任意整数来递增（INCRBY），也可以递减（DECR，DECRBY）。

8.2.2 事务

我们已经在前面的数据库（Postgres 和 Neo4j）中看到过事务，Redis 的 MULTI 块原子命令是类似的概念。如果将两个操作放在一个块内，例如 SET 和 INCR，那么这两个操作要么都成功执行，要么都不执行。永远不会得到部分执行的结果。

我们将在一个事务内，以另一个短码作为一个 URL 的键，并递增计数。我们用 MULTI 命令开始事务，并用 EXEC 命令执行它。

```
redis 127.0.0.1:6379> MULTI
OK
redis 127.0.0.1:6379> SET prag http://pragprog.com
QUEUED
redis 127.0.0.1:6379> INCR count
QUEUED
redis 127.0.0.1:6379> EXEC
1) OK
2) (integer) 2
```

在使用 MULTI 命令时，命令在定义时实际上并不执行（类似于 Postgres 的事务），而是排入队列，然后按顺序执行。

类似于 SQL 中的 ROLLBACK，可以用 DISCARD 命令停止事务，这将清除事务队列。不同于 ROLLBACK，它不会恢复数据库，它只是根本不运行事务。尽管底层的概念是不同的机制（事务回滚与操作取消），但效果是相同的。

8.2.3　复杂数据类型

到目前为止，我们还没有看到太多复杂的行为。用键存储字符串和整数值（甚至是事务），都很好且没有问题，但大多数编程和数据存储的问题需要处理许多数据类型。Redis 能存储列表、哈希表、集合和有序集合，这自然解释了它的受欢迎程度。在探讨了那些可以定制的复杂操作之后，你就会同意这种说法。

采用这些集合数据类型，每个键可以包含大量的值（最多 2^32 个元素，即超过 40 亿）。所有的 Facebook 账户作为一个列表，放在一个键下也毫无问题。

虽然有些 Redis 命令可能看起来有点神秘，但它们一般遵循一个好的模式。SET 命令以 S 开始，哈希表是 H，而有序集合是 Z。列表命令一般以 L（左）或 R（右）开始，这取决于操作的方向（如 LPUSH）。

1. 哈希表

哈希表类似于嵌套的 Redis 对象，可以存储任意数量的键-值对。我们使用一个哈希表来记录用户，他们注册了我们的短 URL 服务。

哈希表很好用，因为它有助于避免使用不自然的键前缀来存储数据。（请注意，在键中用冒号[:]。这是一个有效的字符，通常在逻辑上将键分隔成几段。这仅仅是一个惯例，在 Redis 中没有更深的含义。）

```
redis 127.0.0.1:6379> MSET user:eric:name "Eric Redmond" user:eric:password s3cret
OK
redis 127.0.0.1:6379> MGET user:eric:name user:eric:password
1) "Eric Redmond"
2) "s3cret"
```

不使用分离的键，可以创建一个哈希表，包含它自己的键-值对。

```
redis 127.0.0.1:6379> HMSET user:eric name "Eric Redmond" password s3cret
OK
```

只需要记录单个 Redis 键，就能获取哈希表的所有值。

```
redis 127.0.0.1:6379> HVALS user:eric
1) "Eric Redmond"
2) "s3cret"
```

也可以获取所有哈希键。

```
redis 127.0.0.1:6379> HKEYS user:eric
1) "name"
2) "password"
```

或者，可以通过传递 Redis 键，后面跟哈希键，得到单个值。这里只得到了密码。

```
redis 127.0.0.1:6379> HGET user:eric password
"s3cret"
```

不同于文档型数据存储 MongoDB 和 CouchDB，Redis 的哈希表不能嵌套（任何其他的复杂数据类型也不行，如列表）。换言之，哈希表只能存储字符串值。

还有更多的命令，用于删除哈希字段（HDEL），以某个计数值来递增一个整数字段的值（HINCRBY），或获取一个哈希表中的字段数（HLEN）。

2. 列表

列表包含多个有序值，既可以作为队列（先进先出），也可以作为栈（后进先出）。它们还有更复杂的操作，在列表中的某处插入，限制列表的大小，以及在列表之间移动值。

由于短 URL 服务现在可以记录用户，因此我们希望让它们保存一个愿望列表，记录想访问的 URL。为了创建想访问网站的短码列表，设置键为 USERNAME:wishlist，并将任意数量的值压入列表的右边（末尾）。

```
redis 127.0.0.1:6379> RPUSH eric:wishlist 7wks gog prag
(integer) 3
```

类似于大多数集合值的插入，Redis 命令返回推入值的数量。换言之，将三个值压入列表，所以它返回 3。随时可以用 LLEN 命令来获取列表的长度。

利用列表范围命令 LRANGE，可以指定第一个和最后一个位置，取得列表的任何部分。Redis 中所有列表操作使用从零开始的索引。负的位置是指从末尾算起的步数。

```
redis 127.0.0.1:6379> LRANGE eric:wishlist 0 -1
1) "7wks"
2) "gog"
3) "prag"
```

LREM 命令从给定的键中删除一些匹配的值。它也需要一个数目，以知道要删除多少个匹配的值。这里设定计数为 0，即删除所有匹配的值：

```
redis 127.0.0.1:6379> LREM eric:wishlist 0 gog
```

设定计数大于 0，将只删除这个数目的匹配值。计数设定为负数，将删除该数目的匹配值，但从列表的末尾（右侧）扫描。

如果想按添加的顺序来删除并获取每个值（像队列一样），就可以从列表左侧（头部）弹出它们。

```
redis 127.0.0.1:6379> LPOP eric:wishlist
"7wks"
```

如果想表现得像栈一样，在 RPUSH 值后，要从列表尾部 RPOP。所有这些操作都在常数时间内完成。

关于前面的命令组合，也可以使用 LPUSH 和 RPOP 命令实现类似的效果（队列），或使用 LPUSH 和 LPOP 命令实现栈的效果。

假设我们想从愿望列表中移除值，放入另一个已访问网站的列表中。为了原子地执行这个移动操作，可以将弹出和压入操作放在一个 MULTI 块内。在 Ruby 中，这些步骤看起来可能是这样的（这里不能使用 CLI，因为你必须保存弹出的值，所以我们使用 redis-rb gem）：

```
redis.multi do
  site = redis.rpop('eric:wishlist')
  redis.lpush('eric:visited', site)
end
```

但 Redis 提供了一条命令，从一个列表的尾部弹出值，并压入另一个列表的头部。这称为 RPOPLPUSH（右弹出，左压入）。

```
redis 127.0.0.1:6379> RPOPLPUSH eric:wishlist eric:visited
"prag"
```

如果你查询愿望列表的范围，prag 将消失；现在它在 visited 列表中。这是一个有用的命令排队机制。

如果你查阅 Redis 文档以找到 RPOPRPUSH、LPOPLPUSH 和 LPOPRPUSH 命令，你可能会沮丧地发现它们不存在。RPOPLPUSH 是你唯一的选择，你必须相应地构建你的列表。

3. 阻塞列表

既然短 URL 服务已启动，就添加一些社交活动，比如，添加一个实时评论系统，让人们发布帖子，评论他们已经访问过的网站。

我们来写一个简单的消息传输系统，其中多个客户端可以压入评论，并且一个客户端（整理者）从队列中弹出消息。我们希望整理者仅仅监听新的评论，在它们到达时弹出它们。Redis 为这种目的提供了一些阻塞命令。

首先，打开另一个终端，并启动另一个 redis-cli 客户端。这将是整理者。阻塞直到有值可弹出的命令是 BRPOP。它需要指定弹出值所属的键，以及超时的秒数，设置为 5 分钟。

```
redis 127.0.0.1:6379> BRPOP comments 300
```

然后切换回第一个控制台，把一条消息压入评论里。

```
redis 127.0.0.1:6379> LPUSH comments "Prag is great! I buy all my books there."
```

如果你切换回整理者控制台，将看到两行返回：键和弹出的值。控制台也会输出它花费在阻塞上的时间。

```
1) "comments"
2) "Prag is great! I buy all my books there."
(50.22s)
```

还有一个左弹出的阻塞版本（BLPOP），以及右弹出、左推入的阻塞版本（BRPOPLPUSH）。

4. 集合

短 URL 服务发展顺利，如果能用某种方法将常用的 URL 分组就更好了。

集合是无序聚合，没有重复的值，是两个或两个以上的键值之间执行复杂操作的很好选择，例如，并集或交集等。

如果我们想用一个共同的键将多组 URL 归类，可以使用 SADD 命令添加多个值。

```
redis 127.0.0.1:6379> SADD news nytimes.com pragprog.com
(integer) 2
```

Redis 添加了两个值。可以通过 SMEMBERS 命令获取整个集合，顺序是不确定的。

```
redis 127.0.0.1:6379> SMEMBERS news
1) "pragprog.com"
2) "nytimes.com"
```

我们为技术相关网站添加另一个类别，称为 tech。

```
redis 127.0.0.1:6379> SADD tech pragprog.com apple.com
(integer) 2
```

为找到两个网站集合的交集，也就是既提供新闻又聚焦技术，使用 SINTER 命令。

```
redis 127.0.0.1:6379> SINTER news tech
1) "pragprog.com"
```

可以从一个集合中删除在另一集合中出现的所有值，这也同样容易。要找到所有非技术类的新闻网站，使用 SDIFF 命令：

```
redis 127.0.0.1:6379> SDIFF news tech
1) "nytimes.com"
```

还可以建立一个网站并集，包含新闻网站或技术网站。因为它是一个集合，所以重复元素将丢弃。

```
redis 127.0.0.1:6379> SUNION news tech
1) "apple.com"
2) "pragprog.com"
3) "nytimes.com"
```

这个并集也可以直接存储到一个新的集合中（SUNIONSTORE destination key [key ...]）。

```
redis 127.0.0.1:6379> SUNIONSTORE websites news tech
```

这也提供了一个有用的技巧，将一个键的值复制到另一个键中，如 SUNIONSTORE news_copy news。还有类似的命令，用于存储交集（SINTERSTORE）和差集（SDIFFSTORE）。

就像 RPOPLPUSH 命令将值从一个列表移到另一个列表，SMOVE 命令对集合完成相同的任务，但它更容易记。

就像 LLEN 命令查询列表的长度，SCARD（集合基数）对集合计数，但它更难记。

因为集合是无序的，所以没有左、右或其他位置命令。从集合弹出一个随机值，只需要 SPOP 键，删除值的命令是 SREM key value [value ...]。

不同于列表，集合没有阻塞命令。

5. 有序集合

之前我们看到的其他 Redis 数据类型，很容易映射到常见的编程语言结构上，而有序集合从之前的每个数据类型里取了一些东西。它们像列表一样有序，像集合一样元素唯一。它们像哈希表一样有字段-值对，但没有字符串字段，而是代之以数字，表示表示值的顺序。可以将有序集合想象为一个随机存取的优先级队列。但这种能力也有代价。在内部，因为有序集合保持值的顺序，所以插入的时间复杂度是 log（N）（其中 N 是集合的大小），不像哈希表或列表的时间复杂度是常量。

下面要记录特定短码的流行度。每次有人访问一个 URL 时，得分就会增加。类似于哈希表，在有序集合中增加一个值，需要在 Redis 的键名后跟两个值：得分和成员。

```
redis 127.0.0.1:6379> ZADD visits 500 7wks 9 gog 9999 prag
(integer) 3
```

要增加一个得分，要么重新添加新的得分，这只是更新得分，但没有添加一个新值；要么按某个数字做递增，这将返回新的值。

```
redis 127.0.0.1:6379> ZINCRBY visits 1 prag
"10000"
```

也可以在 ZINCRBY 中设置负数，实现递减。

6. 范围

要从 visits 集合获取值，可以发出一条范围命令，ZRANGE，它按位置返回，就像列表数据类型的 LRANGE 命令。但对于有序集合，位置按得分从最低到最高排序。因此，要获得得分前两位的被访问网站（从零开始），使用下面的命令：

```
redis 127.0.0.1:6379> ZRANGE visits 0 1
1) "gog"
2) "7wks"
```

如果还要得到每个元素的得分，就在前面的命令加上 WITHSCORES。要得到反序的元素，在命令中插入 REV，即 ZREVRANGE。

```
redis 127.0.0.1:6379> ZREVRANGE visits 0 -1 WITHSCORES
1) "prag"
2）"10000"
3) "7wks"
4）"500"
5) "gog"
```

```
6）"9"
```

但是，如果我们正在使用一个有序集合，很可能我们要按得分而不是位置来划定范围。ZRANGEBYSCORE 的语法与 ZRANGE 稍有不同。默认情况下低和高的范围数字是包含在内的，如果要排除一个得分数字，可以在它前面加上左圆括号“(”。因此，下面的命令将返回所有的分数，其中 9≤分数≤9999：

```
redis 127.0.0.1:6379> ZRANGEBYSCORE visits 9 9999
1) "gog"
2) "7wks"
```

但是以下命令将返回 9＜分数≤9999：

```
redis 127.0.0.1:6379> ZRANGEBYSCORE visits (9 9999
1)  "7wks"
```

还可以按正数和负数值划定范围，包括无穷。这将返回整个集合。

```
redis 127.0.0.1:6379> ZRANGEBYSCORE visits -inf inf
```

也可以使用 ZREVRANGEBYSCORE，反序列出它们。

类似于按排名（索引）或得分取得一个范围内的值，ZREMRANGEBYRANK 和 ZREMRANGEBYSCORE 分别按照排名或得分来删除值。

7. 并集

就像集合数据类型一样，可以创建一个目标键，让它包含一个或多个键的并集或交集。这是 Redis 中更为复杂的命令之一，因为它不仅必须要联合键（这是比较简单的操作），而且要合并（可能）不同的分数。并集操作看起来像下面这样：

```
ZUNIONSTORE destination numkeys key [key ...]
  [WEIGHTS weight [weight ...]] [AGGREGATE SUM|MIN|MAX]
```

destination 是要存入的键，key 是一个或多个需要做并集的键。numkeys 就是你即将做并集的键的数目，而 weight 是可选的数字，用来乘以相应键的每一个得分（如果有两个键，就可以有两个权重，以此类推）。最后，aggregate 是处理每个加权得分的可选规则，默认是总和，但也可以在许多得分之间选择最小值或最大值。

将用这条命令来衡量一个短码的有序集合的重要性。

首先，将创建另一个键，记录各个短码的投票得分。站点的每个访问者都可以投票，

表明他们是否喜欢这个网站，每张票增加一个点。

```
redis 127.0.0.1:6379> ZADD votes 2 7wks 0 gog 9001 prag
(integer) 3
```

我们想结合选票和访问量，找出我们系统中最重要的网站。选票很重要，但紧随其后，网站访问量也占一定权重（也许人们对该网站非常着迷，但忘了投票）。我们想要添加两种类型的分数，共同计算出一个新的重要性分数，其中选票具有双倍的重要性，即乘以二。

```
ZUNIONSTORE importance 2 visits votes WEIGHTS 1 2 AGGREGATE SUM
(integer) 3
redis 127.0.0.1:6379> ZRANGEBYSCORE importance -inf inf WITHSCORES
1) "gog"
2)"9"
3) "7wks"
4)"504"
5) "prag"
6)"28002"
```

这个命令的其他用法也很强大。例如，如果需要对集合内的所有成绩加倍，就可以用 2 的权重对单个键做并集操作，并把结果存储回它本身。

```
redis 127.0.0.1:6379> ZUNIONSTORE votes 1 votes WEIGHTS 2
(integer) 2
redis 127.0.0.1:6379> ZRANGE votes 0 -1 WITHSCORES
1) "gog"
2)"0"
3) "7wks"
4)"4"
5) "prag"
6)"18002"
```

对有序集合也有一个类似的命令，用于执行交集操作（ZINTERSTORE）。

8.2.4　到期

像 Redis 这样的键-值对系统有一种常见用法，就是作为数据的快速访问缓存，重新获取或计算这些数据代价高昂。到期功能有助于避免总的键集无限增长，做法是安排 Redis 经过一定的时间就删除一个键-值对。

标记一个键为到期，需要 EXPIRE 命令，一个现有的键，以及以秒计算的存在时间。下面设置一个键，并设置它在 10 秒后到期。我们可以在 10 秒内检查这个键是否存在（EXISTS）并返回 1（真）。如果等待执行，它最终将返回 0（假）。

```
redis 127.0.0.1:6379> SET ice "I'm melting…"
OK
redis 127.0.0.1:6379> EXPIRE ice 10
(integer) 1
redis 127.0.0.1:6379> EXISTS ice
(integer) 1
redis 127.0.0.1:6379> EXISTS ice
(integer) 0
```

设置键和到期时间是如此常用的操作，于是 Redis 提供了一条简捷命令，即 `SETEX`。

```
redis 127.0.0.1:6379> SETEX ice 10 "I'm melting…"
```

可以用 TTL 查询一个键的生存时间。像前面那样设置 `ice` 到期，检查它的 TTL 将返回剩余的秒数。

```
redis 127.0.0.1:6379> TTL ice
(integer) 4
```

在键到期之前的任何时候，都可以通过 `PERSIST key` 消除超时。

```
redis 127.0.0.1:6379> PERSIST ice
```

要标记倒计时到特定的时间，可以用 `EXPIREAT`。它接受一个 Unix 时间戳（自 1970 年 1 月 1 日起的秒数），而不是计算秒数。换言之，`EXPIREAT` 是绝对超时，`EXPIRE` 用于相对超时。

有一个常用技巧，只保留最近用过的键：每当你检索一个值时，更新它的到期时间。这是最近使用（MRU，Most Recently Used）缓存算法，确保你最近使用的键将继续保留在 Redis 中，而被忽视的键将正常到期。

8.2.5 数据库命名空间

到目前为止，我们只与单个命名空间交互。就像 Riak 中的桶（bucket），有时需要通过命名空间将键分隔开。例如，如果你写了一个国际化的键值对存储库，可以在不同的命名空间中存储不同的回应内容。键 `greeting` 可以在德文的命名空间里设置为“guten

tag”，在法文的命名空间里设置为“bonjour”。用户选择语言后，应用程序就从指定的命名空间中获取所有的值。

在 Redis 的术语中，命名空间称为数据库（database），以数字为键。到目前为止，我们一直与默认的命名空间 0（也称为数据库 0）交互。这里设置 `greeting` 为英文的 `hello`。

```
redis 127.0.0.1:6379> SET greeting hello
OK
redis 127.0.0.1:6379> GET greeting
"hello"
```

但是，如果通过 SELECT 命令切换到另一个数据库，该键就不可用了。

```
redis 127.0.0.1:6379> SELECT 1
OK
redis 127.0.0.1:6379[1]> GET greeting
(nil)
```

在这个数据库的命名空间设置一个值，不会影响原来命名空间的值。

```
redis 127.0.0.1:6379[1]> SET greeting "guten tag"
OK
redis 127.0.0.1:6379[1]> SELECT 0
OK
redis 127.0.0.1:6379> GET greeting
"hello"
```

既然所有的数据库都运行在同一服务器实例内，Redis 就允许用 `MOVE` 命令，在不同命名空间之间移动键。下面将 `greeting` 移到数据库 2：

```
redis 127.0.0.1:6379> MOVE greeting 2
(integer) 2
redis 127.0.0.1:6379> SELECT 2
OK
redis 127.0.0.1:6379[2]> GET greeting
"hello"
```

如果针对单个 Redis 服务器运行不同的应用程序，又要允许这些应用程序相互之间交换数据，这个功能就有用了。

8.2.6　更多命令

Redis 有大量的其他操作命令，如重命名键（`RENAME`），确定键值的类型（`TYPE`），

以及删除键值对（`DEL`）。还有痛苦危险的 `FLUSHDB`，它从这个 Redis 数据库中删除所有的键，以及灾难性的命令 `FLUSHALL`，它从所有 Redis 数据库删除所有的键。请查看在线文档，找到 Redis 命令的完整列表。

第 1 天总结

Redis 拥有多种数据类型，并能够执行复杂的查询，这使它超越了标准的键-值对存储库。它可以作为一个栈、队列或优先队列；可以作为对象存储系统（通过哈希表）；甚至可以执行复杂的集合操作，如并集、交集和差集（diff）。它提供了许多原子命令，同时对于那些多步命令，它提供了一种事务机制。它有键到期的内置功能，在作为缓存时，这是有用的。

第 1 天作业

求索

1．查找完整的 Redis 命令文档，包括命令详细信息中以大 O 标记（O(x)）的时间复杂度。

实践

1．安装你喜欢的编程语言驱动程序，连接到 Redis 服务器。在一个事务内插入并递增值。

2．使用你选择的驱动程序，创建一个程序，读取阻塞列表并输出到某个地方（控制台、文件、Socket.io 等），并且创建另一个程序写入相同的列表。

8.3　第 2 天：高级用法，分布

第 1 天，我们介绍了作为数据结构服务器的 Redis。今天，我们将在此基础上介绍 Redis 提供的一些高级功能，如管道、发布-订阅模型、系统配置，以及复制。此外，我们将看到如何创建一个 Redis 集群，快速存储大量数据，并使用先进技术介绍 Bloom 过滤器（Bloom filter）。

8.3.1　一个简单的接口

Redis 有 20 000 行源代码，是一个相当简单的项目。但是，除了代码规模，它还有一个简单的接口，接受我们写在控制台里的每一个字符串。使用 Redis 的原因如图 8-1 所示。



图 8-1　原因

1. telnet

可以不用命令行界面与 Redis 交互，而是利用 telnet，通过 TCP 字符流输入命令，并以回车换行符（CRLF，或\r\n）终止该命令。

```
redis/telnet.sh
$ telnet localhost 6379
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
SET test hello
①+OK
GET test
②$5
hello
SADD stest 1 99
③:2
SMEMBERS stest
④*2
$1
1
$2
99

CTRL-]
```

我们可以看到，输入和我们在控制台提供的一样，但控制台的响应更整洁一点。

① Redis 流在 OK 状态前加了一个+号。

② 在它返回字符串 hello 之前，向它发送$5，这意味着“以下字符串有 5 个字符。”

③ 向测试键添加两个集合元素之后，返回的数字 2 前面有“:”，代表一个整数（两个

值添加成功）。

④ 最后，当我们请求两个元素时，返回的第一行以一个星号和数字 2 开始，意味着有两个复杂的值将返回。接下来的两行如同 hello 字符串，但包含字符串 1，后面是字符串 99。

2. 管道

还可以通过使用 BSD 的 netcat（nc）命令，你可能会发现许多 UNIX 机器上已经安装了这条命令，一次性流入我们自己的字符串。使用 netcat，我们必须明确地以 CRLF 结束一行（telnet 隐含就是这样做的）。echo 命令完成后，我们还要睡眠一秒种，给 Redis 服务器一些时间返回。一些 nc 实现有-q 选项，从而不需要睡眠，但不是所有的 nc 实现都是这样，所以请自由尝试一下。

```
$ (echo -en "ECHO hello\r\n"; sleep 1) | nc localhost 6379
$5
hello
```

可以将命令组织为管道的形式，从而利用这种控制方式的优势，或者在单个请求中流入多条命令。

```
$ (echo -en "PING\r\nPING\r\nPING\r\n"; sleep 1) | nc localhost 6379
+PONG
+PONG
+PONG
```

这可能比每次推入一条命令远为高效，如果有意义，就应该考虑这样做——尤其是在事务中。只是要确保每一条命令以\r\n 结束，这是服务器要求的定界符。

3. 发布-订阅

昨天，我们用列表数据类型实现了一个基本的阻塞队列。我们让数据排队，并能够被阻塞弹出命令读取。利用该队列，实现了一个很基本的发布-订阅模型。可以向该队列推送任何数量的消息，当消息可以读取时，唯一的队列读者会弹出它们。阻塞队列很强大，但有限制。在许多情况下，我们想要行为正好倒过来，即几个订阅者需要读取单一发布者的公告，如图 8-2 所示，发布者向所有订阅者发送消息。Redis 提供了一些专门的发布-订阅（或 pub-sub）命令。

昨天我们用阻塞列表实现了评论机制，接下来我们做一点改进，允许一个用户向多个订阅者发布一条评论（而不是只有一个订阅者）。我们首先让一些订阅者连接到一个键，这在 pub-sub 术语中称为通道（channel）。我们启动两个新的客户端，订阅该评论通道。订阅将导致 CLI 阻塞。

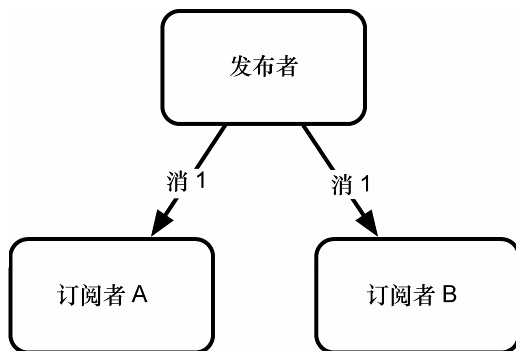


图 8-2　一个发布者向所有订阅者发送一条消息

```
redis 127.0.0.1:6379> SUBSCRIBE comments
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "comments"
3) (integer) 1
```

有了两个订阅者，可以将任何字符串作为一条消息，发布到 comments 通道。PUBLISH 命令将返回整数 2，意思是两个订阅者接收到了它。

```
redis 127.0.0.1:6379> PUBLISH comments "Check out this shortcoded site! 7wks"
(integer) 2
```

两个订阅者都将收到一个多块回复（multibulk reply，是一个列表），包含三个元素：字符串“message”、通道的名字和发布的消息值。

```
1) "message"
2) "comments"
3) "Check out this shortcoded site! 7wks"
```

如果你的客户端希望不再接收到信件，它们可以执行 UNSUBSCRIBE comments 命令，从 comments 通道断开，或者干脆用单独的 UNSUBSCRIBE 命令，从所有通道断开。但是请注意，使用 redis-cli，你需要按 CTRL+C 快捷键来中断连接。

8.3.2　服务器信息

在学习改变 Redis 的系统设置之前，先快速看看 INFO 命令是有价值的，因为更改设置值也将改变这里的某些值。INFO 命令输出服务器数据列表，包括版本、进程 ID、使用的内存和运行时间。

```
redis 127.0.0.1:6379> INFO
redis_version:2.4.5
redis_git_sha1:00000000
redis_git_dirty:0
arch_bits:64
multiplexing_api:kqueue
process_id:54046
uptime_in_seconds:4
uptime_in_days:0
lru_clock:1807217
...
```

在本章中，你可能需要多次使用这个命令，因为它提供了这个服务器全局信息的有用快照和设置。它甚至还提供了持久性、内存碎片和复制服务器状态的信息。

8.3.3 Redis配置

到目前为止，我们只是直接使用 Redis。Redis 的强大之处主要来源于它的可配置性，允许你根据使用情况量身定制。发布版所带的 `redis.conf` 文件，位于*nix 系统的 `/etc/redis` 目录下，该文件基本上是自描述的，所以我们只讨论它的一部分。

我们将依次介绍几个常用设置。

```
daemonize no
port 6379
loglevel verbose
logfile stdout
database 16
```

默认情况下，把 `daemonize` 设置为 no，因此服务器总是在前台启动。这对于测试很好，但对生产环境不是很友好。此值更改为 yes 将在后台运行服务器，同时将服务器的进程 ID 写入一个 pid 文件中。

下一行是这台服务器的默认端口号，端口 6379。如果在一台机器上运行多个 Redis 服务器，这特别有用。`loglevel` 默认设置为 verbose，但在生产环境中将它设置为 `notice` 或 `warning` 会很好。`logfile` 输出到 stdout（标准输出，控制台），但如果在守护进程模式运行 Redis，就需要一个文件名。

`database` 设置我们可用的 Redis 数据库的数量。昨天我们看到了如何在数据库之间切换。如果你打算只使用单个数据库命名空间，将它设置为 1 是不错的。

1. 持久性

Redis 有几个持久性选项。首先是完全不持久，即所有值仅保存在内存中。如果你正在运行一个基本的缓存服务器，这是一个合理的选择，因为持久性总是增加延迟。

与其他类似memcached[1]的快速访问缓存相比，Redis有一点不同，它内置支持将值保存到磁盘。在默认情况下，键值对只是偶尔保存。可以运行LASTSAVE命令，获得Redis最后一次成功写入磁盘的UNIX时间戳，也可以查看服务器INFO输出的last_save_time字段。

可以通过执行 SAVE 命令（或 BGSAVE，在后台异步保存）强制持久。

```
redis 127.0.0.1:6379> SAVE
```

如果你查看 Redis 服务器日志，会看到类似于下面的行：

```
[46421] 10 Oct 19:11:50 * Background saving started by pid 52123
[52123] 10 Oct 19:11:50 * DB saved on disk
[46421] 10 Oct 19:11:50 * Background saving terminated with success
```

另一种持久性方法是在配置文件中改变快照设置。

2. 快照

可以通过添加、删除或改变一个保存字段，来修改存储到磁盘的频率。在默认情况下有 3 项，由关键字 save 作为前缀，后面跟的是以秒计的时间，以及至少几个键发生改变才会写入磁盘。

例如，只要有键改变，每 5 分钟（300 秒）触发一次保存，你会这样写：

```
save 300 1
```

Redis 配置有一套很好的默认值。这组默认值意味着，如果 10 000 个键改变了，在 60 秒内保存；如果改变了 10 个键，在 300 秒内保存；如果改变了 1 个键，至少在 900 秒（15 分钟）内保存。

```
save 900 1
save 300 10
save 60 10000
```

[1] http://www.memcached.org/

可以按照需要增减 save 行，以精确指定阈值。

8.3.4　AOF（append only file）

在默认情况下，Redis 最终是持久的，因为它会根据保存设置定义的时间间隔，将值异步写入磁盘，或根据客户端发起的命令，强制写入磁盘。对于二级缓存或会话服务器，这是可以接受的，但对于存储你需要的持久数据（如金融数据），这是不够的。如果 Redis 服务器崩溃，我们的用户对金钱上的损失可能会不高兴。

Redis 提供了一个仅追加的文件（appendonly.aof），它保留了所有写命令的记录。这类似于我们在第 4 章中看到的预写入日志。如果值还未保存之前服务器崩溃，重新启动时会执行这些命令，恢复其状态；必须在 redis.conf 文件中将 appendonly 设置为 yes 来启用它。

```
appendonly yes
```

然后，我们必须决定命令追加到文件的频率。设置为 always 持久性更好，因为每一条命令都会保存。但它的速度也慢，这与人们使用 Redis 的原因相悖。默认情况下采用 everysec，它节省了时间，每秒钟只写入命令一次。这是一个体面的权衡，因为它足够快，在最坏的情况下，你只会失去最后一秒钟的数据。最后，no 也是一个选项，这只是让操作系统处理磁盘写入。它的写入频率相当低，选择它通常还不如完全忽略仅追加的文件。

```
# appendfsync always
appendfsync everysec
# appendfsync no
```

仅追加的文件有更详细的参数，当你需要应对具体的生产环境问题时，可能值得读一读配置文件里的参数。

1. 安全性

虽然 Redis 本来不打算成为一个完全安全的服务器，但你可能会在 Redis 的文档中遇到 requirepass 设置和 AUTH 命令。忽略它们也没什么问题，因为它们仅仅是设置明文密码的模式。由于客户可能在一秒种内尝试近 10 万个密码，因此它几乎是一个有争议的问题，更不用说明文密码本身就不安全。如果你希望 Redis 安全，最好使用一个好的防火墙和 SSH。

有趣的是，Redis 允许你隐藏或禁止命令，通过晦涩提供命令级别的安全性。下面

将 FLUSHALL 命令（从系统中删除所有的键）改成某个难以猜测的值，如 c283d93ac9528f986023793b411e4ba2：

```
rename-command FLUSHALL c283d93ac9528f986023793b411e4ba2
```

如果试图对这个服务器执行 FLUSHALL，将遇到一个错误。但秘密命令有效。

```
redis 127.0.0.1:6379> FLUSHALL
(error) ERR unknown command 'FLUSHALL'
redis 127.0.0.1:6379> c283d93ac9528f986023793b411e4ba2
OK
```

或者更好的是，可以将它设置为一个空白字符串，完全禁用这条命令。

```
rename-command FLUSHALL ""
```

可以将任意数量的命令设置为空字符串，从而减少命令环境中的命令。

2. 调整参数

有一些更高级的设置，用于加速缓慢的查询日志，编码详细信息，调整延迟，以及导入外部配置文件。但要记住，如果你遇到一些关于 Redis 虚拟内存的文档，你最好尽可能避免它。它已经在 Redis 2.4 弃用了，并可能在将来的版本中删除。

为了帮助测试你的服务器配置，Redis 提供了一个极好的基准测试工具。默认情况下，它连接到本地 6379 端口并使用 50 个并发的客户端发起 10 000 个请求。可以使用参数-n 执行 100 000 个请求。

```
$ redis-benchmark -n 100000
====== PING (inline) ======
  100000 requests completed in 3.05 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

5.03% <= 1 milliseconds
98.44% <= 2 milliseconds
99.92% <= 3 milliseconds
100.00% <= 3 milliseconds
32808.40 requests per second
…
```

还测试了其他命令，如 SADD 和 LRANGE 等；较复杂的命令一般来说会花费更多的时间。

8.3.5　主从复制

就像我们看到过的其他 NoSQL 数据库（如 MongoDB 和 Neo4j），Redis 支持主从复制。如果服务器没有设置为从属服务器，默认就是主服务器。数据将复制到任意数量的从属服务器。

设置从属服务器很容易。首先，需要 `redis.conf` 文件的副本。

```
$ cp redis.conf redis-s1.conf
```

该文件将基本保持不变，只进行以下改动：

```
port 6380
slaveof 127.0.0.1 6379
```

如果一切按计划进行，当启动从属服务器时，应该在从属服务器的日志中看到类似以下的内容：

```
$ redis-server redis-s1.conf

[9003] 16 Oct 23:51:52 * Connecting to MASTER...
[9003] 16 Oct 23:51:52 * MASTER <-> SLAVE sync started
[9003] 16 Oct 23:51:52 * Non blocking connect for SYNC fired the event.
[9003] 16 Oct 23:51:52 * MASTER <-> SLAVE sync: receiving 28 bytes from master
[9003] 16 Oct 23:51:52 * MASTER <-> SLAVE sync: Loading DB in memory
[9003] 16 Oct 23:51:52 * MASTER <-> SLAVE sync: Finished with success
```

你应该看到主服务器日志中字符串 1 `salves` 的输出。

```
redis 127.0.0.1:6379> SADD meetings "StarTrek Pastry Chefs" "LARPers Intl."
```

如果在命令行连接到从属服务器，应该能得到会议列表。

```
redis 127.0.0.1:6380> SMEMBERS meetings
1) "StarTrek Pastry Chefs"
2) "LARPers Intl."
```

在生产环境中，出于可用性或备份目的，你通常希望实现复制，因此让 Redis 从属服务器运行在不同的机器上。

8.3.6　数据转储

到目前为止，关于 Redis 有多快已经谈了很多，但如果不多试一些数据，很难感受到这一点。

我们将一个大型数据集插入Redis服务器。如果你喜欢，你可以保持从属服务器运行，但如果你只有一个主服务器，那么笔记本电脑或台式机可能运行得更快。我们将获取超过 250 万册出版图书的书名列表，键来自于Freebase.com[1]的国际标准图书编号（ISBN）。

首先要安装 redis　Ruby gem。

```
$ gem install redis
```

有几种方法可以插入大型数据集，它们一个比一个快，但一个比一个复杂。

最简单的方法，就是简单地遍历一个数据列表，并对每个值使用标准的 redis-rb 客户端执行 SET 命令。

```
redis/isbn.rb
LIMIT = 1.0 / 0 # 1.0/0 is Infinity in Ruby

# %w{rubygems hiredis redis/connection/hiredis}.each{|r| require r}
%w{rubygems time redis}.each{|r| require r}

$redis = Redis.new(:host => "127.0.0.1", :port => 6379)
$redis.flushall

count, start = 0, Time.now
File.open(ARGV[0]).each do |line|
  count += 1
  next if count == 1
  isbn, _, _, title = line.split("\t")
  next if isbn.empty? || title == "\n"

  $redis.set(isbn, title.strip)

  # set the LIMIT value if you do not wish to populate the entire dataset
  break if count >= LIMIT
end

puts "#{count} items in #{Time.now - start} seconds"
```

[1] http://download.freebase.com/datadumps/latest/browse/book/isbn.tsv

```
$ ruby isbn.rb isbn.tsv
2456384 items in 266.690189 seconds
```

如果你想加快插入速度，并且没有运行 JRuby，可以选择安装 `hiredis` gem。这是一个用 C 写的驱动程序，比原生的 Ruby 驱动程序快很多。然后取消注释 `hiredis` require 行以加载驱动程序。对于这种计算密集型的操作，你可能看不到很大的改善，但我们强烈建议在 Ruby 生产环境中使用 `hiredis`。

利用管道会带来巨大改进。这里以 1000 行作为一个批次，利用管道完成插入。减少的插入时间超过了 300%。

```
redis/isbn_pipelined.rb
BATCH_SIZE = 1000
LIMIT = 1.0 / 0 # 1.0/0 is Infinity in Ruby

# %w{rubygems hiredis redis/connection/hiredis}.each{|r| require r}
%w{rubygems time redis}.each{|r| require r}

$redis = Redis.new(:host => "127.0.0.1", :port => 6379)
$redis.flushall

# set line data as a single batch update
def flush(batch)
  $redis.pipelined do
    batch.each do |saved_line|
      isbn, _, _, title = line.split("\t")
      next if isbn.empty? || title == "\n"
      $redis.set(isbn, title.strip)
    end
  end
  batch.clear
end

batch = []
count, start = 0, Time.now
File.open(ARGV[0]).each do |line|
  count += 1
  next if count == 1

  # push lines into an array
  batch << line

  # if the array grows to BATCH_SIZE, flush it
  if batch.size == BATCH_SIZE
    flush(batch)
```

```
    puts "#{count-1} items"
  end
  # set the LIMIT value if you do not wish to populate the entire dataset
  break if count >= LIMIT
end
# flush any remaining values
flush(batch)

puts "#{count-1} items in #{Time.now - start} seconds"

$ ruby isbn_pipelined.rb isbn.tsv
2666642 items in 79.312975 seconds
```

这减少了所需的 Redis 连接数，但创建管道的数据集本身也有一些开销。如果在生产环境中使用管道，你应该尝试不同数量的批次操作。

对于 Ruby 用户顺便提醒一下，如果应用程序通过 Event Machine 实现非阻塞，Ruby 驱动程序可以通过 `EM::Protocol:: _Redis.connect` 使用 em-synchrony。

8.3.7　Redis集群

除了简单的复制，很多 Redis 客户端提供了一个接口，用于建立一个简单的专用分布式 Redis 集群。Ruby 客户端 redis-rb 支持一致哈希管理的（consistent-hashing managed）集群。你可能还记得第 3 章中的一致哈希，其中节点可以添加和删除，而无须使大多数键到期。这是同样的想法，只通过一个客户端管理，而不需要服务器本身。

首先，需要另一台服务器。不同于主从设置，两个服务器都将采用主服务器（默认）配置。复制 `redis.conf` 文件并修改端口为 6380。这是服务器所需的全部配置。

```
redis/isbn_cluster.rb
LIMIT = 10000

%w{rubygems time redis}.each{|r| require r}
require 'redis/distributed'

$redis = Redis::Distributed.new([
  "redis://localhost:6379/", "redis://localhost:6380/"
])
$redis.flushall

count, start = 0, Time.now
File.open(ARGV[0]).each do |line|
  count += 1
  next if count == 1
```

```
    isbn, _, _, title = line.split("\t")
    next if isbn.empty? || title == "\n"
    $redis.set(isbn, title.strip)

    # set the LIMIT value if you do not wish to populate the entire dataset
    break if count >= LIMIT
  end

  puts "#{count} items in #{Time.now - start} seconds"
```

跨两个或更多的服务器，只需要对现有的 ISBN 客户端做一些小的改动。首先，需要 `require Redis gem` 中的 `redis/distributed` 文件。

```
require 'redis/distributed'
```

然后用 Redis::Distributed 替换 Redis 的客户端，并传入服务器 URI 的数组。每个 URI 需要 `redis` 模式、服务器（localhost）和端口。

```
$redis = Redis::Distributed.new([
  "redis://localhost:6379/",
  "redis://localhost:6380/"
])
```

客户端的运行和以前一样。

```
$ ruby isbn_cluster.rb isbn.tsv
```

但是大量的工作由客户端完成，因为它负责计算哪些键存储在哪个服务器上。可以试着通过 CLI，从每个服务器检索同一个 ISBN 键，从而确认键是存储在单独的服务器上。只有一个客户端能 `GET` 到一个值。但只要通过相同的 Redis::Distributed 配置来检索键集合，客户端将从正确的服务器存取值。

8.3.8 Bloom过滤器

拥有独特的名字是极好的策略，这样就很容易在网上找到。如果你要写一本名为《The Jabbyredis》的书，你几乎可以肯定所有搜索引擎都会链接你。我们写一个脚本，针对 ISBN 目录中的所有书名使用的所有单词，快速检查一个单词是否是唯一的。可以用 Bloom 过滤器（bloom filter）来测试一个单词是否用过。

Bloom 过滤器是一个概率数据结构，它检查一个项是否不在集合中，第一次提到它是在 4.3.4 节。虽然它可能存在误判，但误判不可能不存在。如果你需要迅速发现一个值是否

在系统中不存在，它是非常有用的。

Bloom 过滤器将转换一个值为一个非常稀疏的位序列，与所有值的位序列并集进行比较，从而成功发现不存在性。换言之，当添加一个新值时，它对当前的 Bloom 过滤器位序列执行或（OR）运算。如果要检查一个值是否已经在系统中，就对 Bloom 过滤器的序列执行与（AND）运算。如果该值有一些位为真，但 Bloom 过滤器的对应位不为真，则从未添加该值。换言之，这个值绝对不在 Bloom 过滤器中。这个概念的图形表示，请参阅图 8-3。

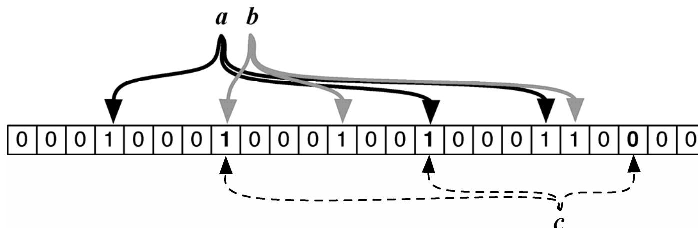


图 8-3　Bloom 过滤器只检查不存在性

我们来写一个程序，对一系列 ISBN 书籍数据循环，提取并简化每本书的书名文字，并将它们分割成单个单词。每个遇到的新词都用 Bloom 过滤器检查。如果 Bloom 过滤器返回 false，表明在 Bloom 过滤器中不存在这个词，然后继续并添加它。就这样一直运行，可以输出所有添加的新词。

```
$ gem install bloomfilter-rb
```

redis/isbn_bf.rb

```
# LIMIT = 1.0 / 0 # 1.0/0 is Infinity in Ruby
LIMIT= 10000

%w{rubygems time bloomfilter-rb}.each{|r| require r}

bloomfilter = BloomFilter::Redis.new(:size => 1000000)

$redis = Redis.new(:host => "127.0.0.1", :port => 6379)
$redis.flushall

count, start = 0, Time.now
File.open(ARGV[0]).each do |line|
  count += 1
  next if count == 1
    _, _, _, title = line.split("\t")
  next if title == "\n"

  words = title.gsub(/[^\w\s]+/, '').downcase
  # puts words
```

```
      words = words.split(' ')

      words.each do |word|
        # skip any keyword already in the bloomfilter
        next if bloomfilter.include?(word)
        # output the very unique word
        puts word
        # add the new word to the bloomfilter
        bloomfilter.insert(word)
      end
      # set the LIMIT value if you do not wish to populate the entire dataset
      break if count >= LIMIT

    end

    puts "Contains Jabbyredis? #{bloomfilter.include?('jabbyredis')}"

    puts "#{count} lines in #{Time.now - start} seconds"
```

Ruby 神童伊利亚·格里高里克（ILya Grigorik）创造了这个基于 Redis 的 Bloom 过滤器，但是这个概念可以用于任何编程语言。

使用相同的 ISBN 文件运行客户端，但是只需要书名。

```
$ ruby isbn_bf.rb isbn.tsv
```

在输出的开始，你应该看到大量常用词，比如，and 和 the。接近集合的结尾，单词越来越深奥难懂，比如，unindustria。

这种方法的好处是能够检测到重复的单词。缺点是会存在一些误报：Bloom过滤器可能误判我们从未见过的一个单词。这就是为什么在一个真实的用例中，你会进行一些辅助检查，如对一个记录系统执行较慢的数据库查询，它应该只在很少的时候发生，假定过滤器的尺寸足够大，这是可计算的[1]。

8.3.9　SETBIT和GETBIT

正如前面提到的，Bloom 过滤器在稀疏二进制字段中翻转某些位，从而发挥作用。在刚才用到的 Redis Bloom 过滤器实现中，使用了两个较新的 Redis 命令来执行这样的操作，即 SETBIT 和 GETBIT。

---

[1] http://en.wikipedia.org/wiki/Bloom_filter

像所有的 Redis 命令一样，SETBIT 具有很好的描述性。该命令在一个位序列的特定位置设置一位（1 或 0），该序列初始为零。这常用于高性能的多元标记：翻转几位要快于写一组描述性字符串。

如果我们要记录汉堡包的配菜，就可以给每个配菜类型分配一个二进制位，如番茄酱 ＝ 0，芥末 ＝1，洋葱 ＝2，生菜 ＝3。因此，只有芥末和洋葱的汉堡包可以表示为 0110，并在命令行设置：

```
redis 127.0.0.1:6379> SETBIT my_burger 1 1
(integer) 0
redis 127.0.0.1:6379> SETBIT my_burger 2 1
(integer) 0
```

稍后，一个过程可以检查我的汉堡是否有生菜或芥末。返回 0 表示没有，返回 1 表示有。

```
redis 127.0.0.1:6379> GETBIT my_burger 3
(integer) 0
redis 127.0.0.1:6379> GETBIT my_burger 1
(integer) 1
```

Bloom 过滤器的实现利用了这一点，它将一个值哈希为多位值。它在 insert() 中对每个为 1 的位置调用 SETBIT X 1（其中 X 是该位的位置），并在 include?() 中调用 GETBIT 验证存在性：如果任何 GETBIT 位置返回 0，返回 false。

如果底层系统较慢，Bloom 过滤器对于减少不必要的流量是极佳的。底层系统可能是较慢的数据库、受限制的资源，或网络请求。如果你有一个较慢的 IP 地址数据库，并且你要跟踪访问网站的所有新用户，就可以先用 Bloom 过滤器检查一个 IP 地址是否存在于系统中。如果 Bloom 过滤器返回 false，你知道 IP 地址尚未添加，并可以作出相应的反应。如果 Bloom 过滤器返回 true，这个 IP 地址在后端可能存在，也可能不存在，需要一定的辅助查找以确定。这就是为什么计算正确的大小是非常重要的：一个大小合适的 Bloom 过滤器可以减少（但不消除）错误率或误判的可能性。

8.3.10　第 2 天总结

今天，我们丰富了对 Redis 的研究，超越了简单的操作，从一个非常快的系统中榨取了最后一点速度。正如我们在第 1 天看到的，Redis 提供了快速、灵活的数据结构存储和简单操作，但是通过内置的发布-订阅功能和位操作，它同样善于完成更复杂的行为。它的可配置性非常好，有许多持久性和复制设置选项，满足你的各种需求。它还支持一些不错的

第三方扩展，如 Bloom 过滤器和集群。

这也结束了 Redis 数据结构存储的主要操作的介绍。明天，我们将要做一些略微不同的事情，以 Redis 作为基石，加上 CouchDB 和 Neo4j，实现多持久并存（polyglot persistence）。

第 2 天作业

求索

1. 找出都有哪些消息模式，并发现 Redis 可以实现多少种。

实践

1. 在将所有快照和仅追加文件设置关闭的情况下，运行 ISBN 填充脚本。然后尝试将 appendfsync 设置为 `always`，再运行脚本，记下速度差异。
2. 使用你最喜爱的编程语言的 Web 框架，尝试构建一个简单的短 URL 服务，使用 Redis 作为后端，提供一个 URL 输入框，并支持基于 URL 的简单重定向。后端使用 Redis 主从复制集群，在多个节点上进行备份。

8.4 第 3 天：与其他数据库合作

今天，我们将结束数据库的最后一章，配合使用前面介绍的一些数据库。当然，Redis 将发挥主要作用，使我们与其他数据库之间的互动更快，更容易。

在整本书中，我们已经了解到，不同的数据库有各自不同的优势，有许多现代系统的设计转向多持久并存的模型，各种数据库在系统中各自发挥不同的作用。你将学习如何构建这样的一个项目，用 CouchDB 作为记录系统（规范的数据源），用 Neo4j 处理数据关系，并用 Redis 辅助实现数据填充和缓存。请把这个项目看成你的期末考试。

请注意，这个项目并不证明本书的作者偏爱一套特定的数据库、语言或框架，而是为了展示多个数据库可以如何配合，充分利用各自的能力，达到一个共同的目标。

8.4.1 多持久并存服务

多持久并存服务将作为乐队信息服务的一个前端。我们要存储乐队名称的列表，在这些乐队中表演的艺术家，以及每个艺术家在乐队中担任的各种角色，从主唱到后备键盘吉他演奏员等。三个数据库（Redis、CouchDB 和 Neo4j）分别处理乐队管理系统的不同方面。

Redis 在系统中起着三个重要的作用：协助将数据填充到 CouchDB，作为 Neo4j 最近

变化的缓存，作为部分值搜索的快速查找。它的速度和存储多种数据格式的能力非常适合数据填充，而其内置的到期策略可以完美地处理数据缓存。

CouchDB 是记录系统（SOR，System Of Record）或权威数据源。CouchDB 的文档结构能够方便地存储乐队数据，包括嵌套的艺术家和角色信息，同时将利用 CouchDB 中的 Changes API，保持第三个数据源的同步。

Neo4j 是关系存储库。虽然直接查询 CouchDB 的 SOR 是完全合理的，但图形数据存储库允许我们简单快速地通过节点关系导航，而其他的数据库在时间上难以匹敌。我们将存储乐队、乐队成员，以及成员扮演的角色之间的关系。

每个数据库都在系统中发挥特定的作用，但它们不会原生地通信。我们使用 Node.js JavaScript 框架填充数据库，在它们之间通信，并作为一个简单的前端服务器。由于将多个数据库联接在一起需要一些代码，所以在这最后一天，我们看到的代码比以前多得多。

8.4.2 数据填充

第一项业务是用必要的数据填充数据存储。这里采取两阶段的方法，首先填充 Redis 数据库，然后填充 CouchDB SOR。

多持久并存的兴起

多语言编程的现象正日益增多，同样，多持久并存正不断取得进展。

如果你不熟悉这种做法，多语言编程是一个团队在单个项目中使用多种编程语言。与此不同，过去的惯例是在整个项目中使用一种通用语言。使用多种编程语言是有用的，因为不同的语言有其固有优势。像 Scala 这样的框架，可能更适合在 Web 上处理服务器端的无状态事务，而像 Ruby 这样的语言可能对业务逻辑更友好。一起使用时，它们形成了配合。众所周知，Twitter 就使用了这样的多种语言系统。我们看到的一些数据库本身就支持多语言编程，Riak 在编写 mapreduce 时支持 JavaScript 和 Erlang，而且一个请求就可以执行这两种语言。

类似于多语言编程，使用多持久共存，你可以在同一系统中利用多种数据库的优势，而不是目前熟悉的方式，即使用单一数据库，很可能是关系数据库。这种形式的一个基本变异已经很常见了：使用 Redis 这样的键值对存储库，作为较慢的关系数据库（如 PostgreSQL）查询的缓存。正如我们在前面的章节中看到的，关系数据库的一些问题并非最佳选择，如图的遍历。这样的问题越来越多。但即使是这些新数据库，也只是整个需求银河中闪耀的几颗星星。

> 为什么突然对多持久共存产生了兴趣？马丁 · 福勒（Martin Fowler）曾经写道[a]，利用一个中央数据库来集成多个应用程序，这是软件设计中常见的模式。这个数据库集成模式一度很流行，现在已经让位于中间件层模式，即多个应用程序通过基于 HTTP 的服务层通信。这让中间件服务不必依赖于任何数量的数据库，在多持久并存的情况下，不必依赖于任何类型的数据库。
>
> a. http://martinfowler.com/bliki/DatabaseThaw.html

如同在前面的章节中一样，从 Freebase.com 下载数据集。这里将使用 group_membership，它是用制表符分隔的数据集[1]。该文件包含了大量信息，但是我们只有兴趣提取member（艺术家的名字）、group（乐队的名字），以及roles（他们在乐队中扮演的角色，另存为一个逗号分割的列表）。例如，John Cooper在Skillet乐队中是Lead vocalist、Acoustic guitar player和Bassist.（主唱、声学吉他手和贝斯手。）

```
/m/0654bxy John Cooper Skillet Lead vocalist,Acoustic guitar,Bass 1996
```

最终，要将 John Cooper 和 Skillet 乐队的其他成员组织到下面这样一个 CouchDB 文档中，存储于 URL http://localhost:5984/bands/Skillet：

```
{
  "_id": "Skillet",
  "name": "Skillet"
  "artists": [
    {
      "name": "John Cooper",
      "role": [
        "Acoustic guitar",
        "Lead vocalist",
        "Bass"
      ]
    },
    ...
    {
      "name": "Korey Cooper",
      "role": [
        "backing vocals",
        "Synthesizer",
        "Guitar",
        "Keyboard instrument"
      ]
    }
  ]
}
```

[1] http://download.freebase.com/datadumps/latest/browse/music/group_membership.tsv

这个文件包含了超过 10 万名乐队成员，以及超过 3 万个乐队。这不是很多，但它是一个很好的起点，用于建立你自己的系统。请注意，并非每个艺术家的角色有记录。这是一个不完整的数据集，但我们可以稍后再处理这个问题。

1. 第 1 阶段：数据转换

你可能想知道，为什么我们要费事地填充 Redis，而不是直接填充 CouchDB。Redis 作为中间人，为扁平的 TSV 数据添加了结构，这样，随后插入另一个数据库就很快。因为我们的计划是为每个乐队创建一条记录，Redis 让我们能够一次扫描 TSV 文件（其中列出了每个乐队成员的名字——每个乐队成员由一行表示）。针对文件中的每一行，直接将一个成员添加到 CouchDB 中，可能导致更新颠簸：两行乐队成员试图同时创建/更新同一乐队文档，当其中一个检查 CouchDB 的版本出现故障时，将迫使系统重新插入。

这一策略有个缺点，它受到 Redis 的限制，只能将整个数据集保存在内存中——但是利用我们在第 2 天看到的简单一致散列集群，这个缺点可以克服。

拿到数据文件后，请确保你安装了 Node.js 以及 Node 程序包管理器（Node Package Manager，NPM）。一旦完成了这些准备工作，就需要安装三个 NPM 项目：redis、csv 和 hiredis（我们昨天学过的可选的 Redis C 驱动程序，可以大大加速 Redis 的交互）。

```
$ npm install hiredis redis csv
```

然后，检查你的 Redis 服务器运行在默认的 6379 端口上，或改变每个脚本的 createClient()函数，指向你的 Redis 端口。

可以在 TSV 文件所在的目录下，运行下面的 Node.js 脚本，向 Redis 填充数据，假设 TSV 文件命名为 `group_membership.tsv`。（我们要看的所有 JavaScript 文件都相当冗长，所以本书没有把它们完整地列出来。所有代码都可以从 Pragmatic Bookshelf 的网站下载。这里我们只关注每个文件的核心内容。）下载并运行以下文件：

```
$ node pre_populate.js
```

这个脚本主要是遍历 TSV 文件的每一行并提取艺术家的名字、乐队的名字，以及艺术家在乐队中扮演的角色。然后，它将这些值加到 Redis（跳过所有空值）。

每个 Redis 乐队键的格式是`"band:乐队的名字"`。该脚本将这个艺术家的名字添加到艺术家名字的集合中。所以，键`"band:Beatles"`将包含值的集合`["John Lennon", "Paul McCartney", "George Harrison", "Ringo Starr"]`。艺术家键也将包含乐队的名字并同样包含角色的集合。`"artist:Beatles:Ringo Starr"` 将包含集合`["Drums"]`。

其他的代码只是记录我们已经处理了多少行，并将结果输出到屏幕上。

redis/pre_populate.js
```
csv().
fromPath( tsvFileName, { delimiter: '\t', quote: '' }).
on('data', function(data, index) {
  var
    artist = data[2],
    band = data[3],
    roles = buildRoles(data[4]);

  if( band === '' || artist === '' ) {
    trackLineCount();
    return true;
  }

  redis_client.sadd('band:' + band, artist);
  roles.forEach(function(role) {
    redis_client.sadd('artist:' + band + ':' + artist, role);
  });
  trackLineCount();
}).
```



埃里克说：

非阻塞代码

在开始这本书之前，我们只是顺便熟悉了编写事件驱动的非阻塞应用程序。非阻塞的意思很明确：不等待一个长时间运行的过程完成，主代码将继续执行。不论你响应阻塞事件需要做什么，都放在一个函数或代码块内，以后再执行。实现方式可以是生成一个单独的线程，也可以像该例子，实现一个反应器模式（reactor pattern）事件驱动方法。

在阻塞程序中，可以编写代码查询数据库、等待，并遍历结果。

```
results = database.some_query()
for value in results
  # do something with each value
end
# this is not executed until after the results are looped...
```

在事件驱动程序中，你将一个循环作为一个函数或代码块传入。当数据库正在做它自己的事情时，程序的其余部分可以继续运行。只有在数据库返回结果时，函数/代码块才执行。

```
database.some_query do |results|
  for value in results
    # do something with each value
  end
end
# this continues running while the database performs its query...
```

我们花了很长一段时间，才认识到这样做的好处。当等待数据库时，程序的其余部分可以运行而不是闲置，确实如此，但是这常见吗？显然是这样，因为当我们开始以这种风格编码时，我们注意到延迟下降了一个数量级。

我们尽可能保持代码的简单，可是以非阻塞方式与数据库交互本来就是一个复杂的过程。但正如我们所了解到的，一般来说，这是与数据库打交道的一个很好的方法。几乎每一种流行的编程语言都有某种非阻塞库。Ruby 有 EventMachine，Python 有 Twisted，Java 提供了 NIO 库，C # 有 Interlace，当然，JavaScript 有 Node.js。

通过启动 redis-cli 并执行 RANDOMKEY，可以检查该代码已经填充了 Redis。我们预期会看到一个带前缀 band: 或 artist: 的键，值不是（nil）就对了。

既然 Redis 已经被数据填充，立即着手下一步。关闭 Redis 可能会丢失数据，除非你选择设置比默认更高级别的持久性，或启动 SAVE 命令。

2. 第 2 阶段：SOR 插入

CouchDB 将作为记录系统（SOR）。如果在 Redis、CouchDB 或 Neo4j 之间发生任何数据冲突，CouchDB 将胜出。一个良好的 SOR 应该包含所有必要的数据，以便在需要时重建该领域中的任何其他的数据源。

确保 CouchDB 运行在默认的 5984 端口上，或在下面的代码中把 require('http').createClient(5984,'localhost')将更改为你需要的端口号。Redis 也应该仍然运行着。下载并运行以下文件：

```
$ node populate_couch.js
```

因为第 1 阶段所做的都是从 TSV 获取数据并填充 Redis，所以这个阶段所做的是获取 Redis 的数据并填充 CouchDB。我们不使用 CouchDB 的任何特殊驱动程序，因为它是一个简单的 REST 接口，而且 Node.js 有一个内置的简单 HTTP 库。

在下面的代码块中，执行 Redis 的命令 KEYS bands:*，得到在系统中所有的乐队名字的列表。如果我们有一个非常大的数据集，我们可以加上更小的范围限定（例如，bands:A* 只得到以 a 开头的乐队的名字，等等）。然后，针对每个乐队，获取艺术家的集合，并从键

字符串中删除前缀 bands:，提取键中乐队的名字。

```
redis/populate_couch.js
redisClient.keys('band:*', function(error, bandKeys) {
  totalBands = bandKeys.length;
  var
    readBands = 0,
    bandsBatch = [];

  bandKeys.forEach(function(bandKey) {
    // substring of 'band:'.length gives us the band name
    var bandName = bandKey.substring(5);

    redisClient.smembers(bandKey, function(error, artists) {
```

下一步，我们得到每一位艺术家在这个乐队中的所有角色，Redis 将其作为一个数组的数组（每个艺术家的角色是自己的数组）返回。可以将 Redis 的一些 `SMEMBERS` 命令存入到一个名为 `roleBatch` 的数组中，并在一个 `MULTI` 批次中执行它们，来做到这一点。实际上，这将执行单一的管道化请求，像下面这样：

```
MULTI
  SMEMBERS "artist:Beatles:John Lennon"
  SMEMBERS "artist:Beatles:Ringo Starr"
EXEC
```

从这些数据，生成每批 50 个 CouchDB 文档。建立 50 个文档的批次，因为随后发送整个集合给 CouchDB 的`/_bulk_docs` 命令，让我们非常、非常快速地插入数据。

```
redis/populate_couch.js
redisClient.
  multi(roleBatch).
  exec(function(err, roles)
  {
    var
      i = 0,
      artistDocs = [];

    // build the artists sub-documents
    artists.forEach( function(artistName) {
      artistDocs.push({ name: artistName, role : roles[i++] });
    });

  // add this new band document to the batch to be executed later
```

```
  bandsBatch.push({
    _id: couchKeyify( bandName ),
    name: bandName,
    artists: artistDocs
  });
```

填充完乐队数据库，现在系统要求的所有数据都在一个位置。我们知道许多乐队的名字，在乐队中演出的艺术家，以及他们在这些乐队中扮演的角色。

现在可以好好休息一会，尝试我们刚刚在 CouchDB 中用数据填充的乐队记录系统，地址是 `http://localhost:5984/_utils/database.html?bands`。

8.4.3 关系存储

下一步是 Neo4j 服务，我们将用它来记录艺术家和他们所扮演的角色之间的关系。我们当然可以通过创建视图直接查询 CouchDB，但是我们在基于关系的复杂查询上相当有限。如果 Flaming Lips 乐队的 Wayne Coyne 在表演之前丢失了他的电子琴，他可以向 Nine Inch Nails 乐队的 Charlie Clouser 要，后者也演奏电子琴。或者我们可以发现有很多重叠才能的艺术家，即使他们在不同的乐队中扮演不同的角色——简单遍历各节点就可以做到这一点。

有了初始数据，如果记录系统有任何数据改变，我们都需要保持 Neo4j 与 CouchDB 同步。所以，就采用一石二鸟的办法，建立一个服务，将自数据库创建以来 CouchDB 的所有改变填充到 Neo4j。我们也希望用我们的乐队、艺术家和角色的键填充 Redis，这样以后就可以快速访问这些数据。令人高兴的是，这包括了我们已经填充到 CouchDB 的所有数据，因此我们省掉了单独的 Neo4j 和 Redis 的初始数据填充步骤。

请确保 Neo4j 运行在 7474 端口上，或相应改变 `createClient()` 函数以使用正确的端口。CouchDB 和 Redis 应该保持运行。下载并运行以下文件。这个文件将持续运行直到你关闭它。

```
$ node graph_sync.js
```

这个服务器只是使用在第 6 章中看到的连续投票的例子，追踪所有 CouchDB 的变更。每当检测到更改时，我们做两件事：填充 Redis 并填充 Neo4j。这部分代码通过层叠回调函数填充 Redis。首先，它填充乐队为`"band-name:Band Name"`。艺术家的名字和角色也遵循同一模式。

这样，可以搜索部分字符串。例如，`KEYS band-name:Bea*` 可以返回：Beach Boys、Beastie Boys、Beatles 等。

```
redis/graph_sync.js
function feedBandToRedis(band) {
  redisClient.set('band-name:' + band.name, 1);
  band.artists.forEach(function(artist) {
    redisClient.set('artist-name:' + artist.name, 1);
    artist.role.forEach(function(role){
      redisClient.set('role-name:' + role, 1);
```

下一个代码块是如何填充 Neo4j。创建了一个驱动程序，它是本书中代码的一部分，可以下载它，命名为 neo4j_caching_client.js。它只是使用 Node.js 的 HTTP 库连接到 Neo4j 的 REST 接口，使用了一点儿内置的限速，所以客户端不会一次打开太多的连接。驱动程序还用 Redis 记录了对 Neo4j 的图所做的变更，而无须启动一个单独的查询。这是我们第三次独立使用 Redis——第一次是数据转换，作为填充 CouchDB 的步骤，而第二次是我们刚刚在前面看到的，快速搜索乐队的值。

这部分代码创建了乐队节点（如果需要创建它们），然后是艺术家节点（如果需要创建它们），然后是角色。过程中的每一步创建了一个新的关系，因此 Beatles 节点将与 John，Paul、George 和 Ringo 节点相关，这些节点各自又与他们所扮演的角色相关。

```
redis/graph_sync.js
function feedBandToNeo4j(band, progress) {
  var
    lookup = neo4jClient.lookupOrCreateNode,
    relate = neo4jClient.createRelationship;

  lookup('bands', 'name', band.name, function(bandNode) {
    progress.emit('progress', 'band');
    band.artists.forEach(function(artist) {
      lookup('artists', 'name', artist.name, function(artistNode){
        progress.emit('progress', 'artist');
        relate(bandNode.self, artistNode.self, 'member', function(){
          progress.emit('progress', 'member');
        });
        artist.role.forEach(function(role){
          lookup('roles', 'role', role, function(roleNode){
            progress.emit('progress', 'role');
            relate(artistNode.self, roleNode.self, 'plays', function(){
              progress.emit('progress', 'plays');
```

让这个服务运行在它自己的窗口。每次对 CouchDB 的更新，增加了一个新的艺术家，或为原有的艺术家新增一个角色，都将触发生成 Neo4j 中的新关系，也可能生成 Redis 中的新键。只要这个服务在运行，它们就应该是同步的。

打开 CouchDB Web 控制台，并打开一个乐队。对数据库做出你希望的任意数据变更：添加一个新的乐队成员（使你自己成为披头士乐队的成员！），或者为艺术家添加一个新的角色。注意 graph_sync 的输出。然后启动 Neo4j 的控制台，尝试寻找图中任何新的连接。如果你添加了一个新的乐队成员，现在应该有一个到乐队节点的关系，如果修改，就会有到新角色的关系。当前的实现不会删除关系——尽管不在脚本中添加 Neo4j 的 DELETE 操作就不是完整的修改。

8.4.4 服务

我们终于已经到了这一部分。我们将创建一个简单的 Web 应用程序，允许用户搜索一个乐队。系统中的任何乐队将以链接形式列出所有的乐队成员，而点击任何乐队成员链接将列出一些有关该艺术家的信息——即他们所扮演的角色。此外，艺术家所扮演的每个角色将列出系统中所有其他也扮演过该角色的艺术家。

例如，搜索 Led Zeppelin 会给出 Jimmy Page、John Paul Jones、John Bonham 和 Robert Plant。点击 Jimmy Page 将列出他弹吉他以及很多其他弹吉他的艺术家，如 U2 乐队中的 The Edge。

为了让 Web 应用程序简单，需要两个其他的 node 程序包：bricks（一个简单的 web 框架）和 mustache（一个模板库）。

```
$ npm install bricks mustache
```

和前几节一样，确保你所有的数据库在运行，然后启动服务器。下载并运行以下代码：

```
$ node band.js
```

服务器设置在 8080 端口上运行，因此，如果你将浏览器指向 http://localhost:8080/，你应该看到一个简单的搜索表单。

我们来看看代码，它将建立一个网页，列出乐队的信息。每个 URL 执行在小型 HTTP 服务器中的一个单独的函数。首先是 http://localhost:8080/band，接受任何乐队的名字作为参数。

```
redis/bands.js
appServer.addRoute("^/band$", function(req, res) {
  var
    bandName = req.param('name'),
    bandNodePath = '/bands/' + couchUtil.couchKeyify( bandName ),
    membersQuery = 'g.V[[name:"'+bandName+'"]]'
```

```
                + '.out("member").in("member").uniqueObject.name';

getCouchDoc( bandNodePath, res, function( couchObj ) {
  gremlin( membersQuery, function(graphData) {
    var artists = couchObj && couchObj['artists'];
    var values = { band: bandName, artists: artists, bands: graphData };

    var body = '<h2>{{band}} Band Members</h2>';
    body += '<ul>{{#artists}}';
    body += '<li><a href="/artist?name={{name}}">{{name}}</a></li>';
    body += '{{/artists}}</ul>';
    body += '<h3>You may also like</h3>';
    body += '<ul>{{#bands}}';
    body += '<li><a href="/band?name={{.}}">{{.}}</a></li>';
    body += '{{/bands}}</ul>';

    writeTemplate( res, body, values );
```

如果你在搜索表单中输入乐队 Nirvana，你的 URL 请求将是 `http://localhost:8080/band?name=Nirvana`。该函数将生成一个 HTML 页面（整体模板在一个外部文件中，名为 `template.html`）。此网页列出了一个乐队中的所有艺术家，它直接从 CouchDB 抽取数据。它还列出了一些建议的乐队，是对 Neo4j 图做 Gremlin 查询得到的。对于 Nirvana，Gremlin 查询是：

```
g.V.filter{it.name=="Nirvana"}.out("member").in("member").dedup.name
```

换言之，从 Nirvana 节点出发，得到所有唯一的名字，它们的成员连到 Nirvana 成员。例如，Dave　Grohl 既在 Nirvana 乐队，也在 Foo Fighters，所以 Foo Fighters 将在此列表中返回。

下一个动作是 URL `http://localhost:8080/artist`。该网页将输出关于一个艺术家的信息。

```
redis/bands.js
appServer.addRoute("^/artist$", function(req, res) {
  var
    artistName = req.param('name'),
    rolesQuery = 'g.V[[name:"'+artistName+'"]].out("plays").role.uniqueObject',
    bandsQuery = 'g.V[[name:"'+artistName+'"]].in("member").name.uniqueObject';

  gremlin( rolesQuery, function(roles) {
    gremlin( bandsQuery, function(bands) {
```

```
    var values = { artist: artistName, roles: roles, bands: bands };

    var body = '<h3>{{artist}} Performs these Roles</h3>';
    body += '<ul>{{#roles}}';
    body += '<li>{{.}}</li>';
    body += '{{/roles}}</ul>';
    body += '<h3>Play in Bands</h3>';
    body += '<ul>{{#bands}}';
    body += '<li><a href="/band?name={{.}}">{{.}}</a></li>';
    body += '{{/bands}}</ul>';

    writeTemplate( res, body, values );
```

这里执行了两个 Gremlins 查询。首先输出一个成员承担的所有角色，其次是一个艺术家所在的乐队的列表。例如，Jeff Ward（`http://localhost:8080/ artist?name=Jeff%20Ward`）将作为鼓手，列在 Nine Inch Nails 和 Ministry 乐队中。

前两页有一个很酷的功能：我们展现了这些值之间的联系。`/bands` 页中列出的艺术家链接到选定的`/artist` 页，反之亦然。但我们可以让搜索更容易一些。

```
redis/bands.js
appServer.addRoute("^/search$", function(req, res) {
  var query = req.param('term');

  redisClient.keys("band-name:"+query+"*", function(error, keys) {
    var bands = [];
    keys.forEach(function(key){
      bands.push(key.replace("band-name:", ''));
    });
    res.write( JSON.stringify(bands) );
    res.end();
```

如前面所述，这里只是从 Redis 中抽取匹配字符串第一部分的所有键，如前面描述的"Bea*"。然后以 JSON 格式输出数据。`template.html` 文件链接到必要的 jQuery 代码，使这个功能成为生成的搜索框上的输入自动补全功能。

扩展该服务

对于这里的所有主要工作来说，这是一个相当小的脚本。你可能会发现很多地方你想扩展。请注意，乐队推荐的只是一阶乐队（目前成员曾演出过的乐队）；可以编写一个查询遍历二阶乐队，得到有趣的结果，像这样：`g.V.filter{it.name=='Nine Inch Nails'}.out('member').in('member').dedup.loop(3){ it.loops <=`

2 }.name。

你可能还注意到，我们没有一个可以更新乐队信息的表单。添加该功能相当简单，因为我们已经在 populate_couch.js 脚本中写了 CouchDB 的数据填充代码，只要 graph_sync.js 服务在运行，填充 CouchDB 会自动保持 Neo4j 与 Redis 的最终一致。

如果你喜欢这种多持久共存，甚至可以更进一步。可以添加一个PostgreSQL数据仓库[1]，转化数据为星型模式——允许不同维度的分析，如最常用的演奏乐器或一个乐队中成员总数对总乐器的平均数。可以添加一个Riak服务器存储乐队音乐的样品；添加一个HBase服务器以构建一个消息系统，其中用户可以保存其喜欢/不喜欢的历史；或添加一个MongoDB扩展，为这项服务添加地理元素。

或者，使用完全不同的语言、Web 框架或数据集，重新设计这个项目。数据库和创建它的技术有多少种组合，就有多少扩展这个项目的机会：所有开源技术的笛卡尔乘积。

8.4.5 第 3 天总结

今天是一个大日子——事实上，很大，如果它花了好几天才能完成，我们将不会感到惊讶。但是，这里有点未来数据管理系统的味道，因为世界正在远离一个大型关系数据库的模型，转向几个专门数据库的模型。我们还用一些非阻塞的代码将这些数据库联接在一起，虽然这不是这本书的重点，但似乎也是数据库交互在开发领域的发展方向。

在这个模型中，Redis 的重要性不容忽视。Redis 提供的功能这些数据库肯定都能独自提供，但它确实提供了快速的数据结构。我们能够将一个平面文件组织成一系列有意义的数据结构，这是数据填充和转移中不可分割的一部分。它做这件事的方法既快速又易用。即使你还不能接受多持久共存的模型，你也肯定应该在所有系统中考虑采用 Redis。

第 3 天作业

实践

1. 改变导入步骤，同时记录一个乐队成员与乐队的开始和结束日期。在艺术家的 CouchDB 的子文档中记录该数据。在艺术家的页面上显示此信息。

[1] http://en.wikipedia.org/wiki/Data_warehouse

2. 在组合中添加 MongoDB，将一些音乐样本存入 GridFS，其中用户可以听一两首与乐队相关的歌曲。如果一个乐队有歌曲存在，就在 Web 应用程序上添加链接。确保 Riak 数据与 CouchDB 保持同步。

8.5 总结

Redis 的键-值对（或数据结构）存储是轻量级和紧凑的，具有多种用途。这类似于瑞士军刀，有小刀、开罐器、螺丝锥，及其他各种工具——Redis 很适合完成各种奇怪的任务。总之，Redis 快速、简便，其持久性取决于你的选择。虽然它不是一个独立的数据库，但 Redis 是所有多持久并存生态系统的完美补充，可以作为一个永远存在的帮手，用于转换数据、缓存请求，或通过它的阻塞命令方式来管理消息。

8.5.1 Redis的优点

像许多同类的键值对存储库一样，Redis 的明显优势是速度快。但与大多数键-值对存储库不同的是，Redis 提供存储复杂值的能力，如列表、哈希表和集合，并基于这些数据类型上的特定操作来获取数据。但是，Redis 不只是数据结构存储库，它的持久性选项允许你为了数据安全性而牺牲速度，这到了相当细致的程度。内置的主从复制机制提供了另一种很好的方式，来确保更好的持久性，而无须放慢速度，在每次操作时同步磁盘上的仅追加文件。此外，复制机制对于大量读取的系统是很好的。

8.5.2 Redis的缺点

Redis 很快，在很大程度上是因为它驻留在内存中。有些人可能会认为这是作弊，因为从来不接触磁盘的数据库肯定很快。内存数据库有一个固有的持久性问题，如果你在快照发生之前关闭数据库，你可能会丢失数据。即使你设置在每个操作时都同步磁盘上的仅追加文件，你也冒着回放过期值的风险，因为基于时间的事件永远不会以完全同样的方式重放——虽然公平地说，这种情况是假设多于实际。

Redis 也不会支持比你的可用内存更大的数据集（Redis 将不支持虚拟内存），所以它的大小有实际限制。目前虽然正在开发一个 Redis 集群，目标是超过单机的 RAM 限制，但现在想用 Redis 集群的人必须自己处理，使用支持它的客户端（像我们在第 2 天使用过的 Ruby 驱动程序）。

8.5.3　结束语

Redis 中有大量命令——120 多条。大多数命令简单易懂，名符其实，只要你习惯理解看似随意省略的字母（例如，INCRBY），并理解数学的精确性有时更令人疑惑而不是有帮助（例如，ZCOUNT 表示有序集合计数，SCARD 表示集合基数）。

Redis 已经成为许多系统的组成部分。一些开源项目依赖于 Redis，如 Resque 是一个基于 Ruby 的异步工作排队服务，又如 Node.js 的会话管理项目 SocketStream。无论选择何种数据库作为你的 SOR（记录系统），当然都应该在这个组合中添加 Redis。

第 9 章

结束语

现在，我们已经介绍完了 7 种数据库，祝贺大家！

我们希望你已经了解了这 7 种数据库。如果你在项目中使用某一种数据库，我们会很高兴。如果你决定使用多种数据库，就像我们在第 8 章的结尾看到的，我们会欣喜若狂。我们相信，数据管理的未来在于多持久并存模型（在一个项目中使用多种数据库）——而通用 RDBMS 世界观的迷雾将随风飘逝。

我们借此机会看一看，所介绍的 7 种数据库在更大的数据库生态系统中如何结合在一起。从这一点来说，我们已经探索了每一种数据库的细节，并提到了一些共性和差异。我们将看到它们如何为宏大的、正在扩展的全部数据存储可选方案做出贡献。

9.1 类型终极版

我们已经看到，数据库存储数据的方式可以主要分为 5 种类型：关系型、键-值型、列型、文档型和图型。我们来花点时间，回顾一下它们的区别，看看每种风格适合什么，不太适合什么，也就是说，什么情况下你想使用它们，什么情况下想避免它们。

9.1.1 关系型

这是最常见的经典的数据库模式。关系数据库管理系统（RDBMS），是基于集合理论的系统，实现方式是具有行和列的二维表。关系数据库严格强制使用类型，一般分为数值、字符串、日期和未解释的二进制大对象，但我们看到 PostgreSQL 提供了一些扩展，如数组和 cube。

1. 适合

因为关系数据库的结构性质，如果提前知道数据的布局，但是可能不清楚随后你打算如何使用这些数据，那么关系型数据库是合适的。或者，换句话说，你提前为组织的复杂性付出代价，以实现随后的查询灵活性。许多业务问题正好是以这种方式建模的，从接单到出货以及库存到购物车。你可能事先不知道以后将如何查询数据（例如，我们在 2 月份处理了多少订单？），但数据在本质上是相当规范的，所以强制这种规范性是很有帮助的。

2. 不那么适合

如果你的数据是高度可变的或者多层次的，那么关系数据库不是最合适。因为你必须提前指定模式，所以，处理记录与记录之间有很大变化的数据问题将遇到麻烦。假设考虑开发一个数据库来描述所有自然界中的生物。创建你需要考虑到的所有特征的完整列表（hasHair、numLegs、laysEggs 等）会很棘手。在这种情况下，你选择的数据库最好对可能的输入有较少的预先限制。

9.1.2　键-值存储库

键-值（KV）存储库是我们介绍过的最简单的模型。KV 将简单的键映射到（可能）更复杂的值，就像一个巨大的哈希表。由于它们相对简单，因此这种类型的数据库实现起来最灵活。哈希查找速度快，在 Redis 的例子中就是这样，速度是其主要的关注。哈希查找也容易分布化，所以 Riak 利用这一事实，侧重于简单管理的集群。当然，它的简单性可能对有复杂的建模需求的数据是个缺点。

1. 适合

由于很少或不需要维护索引，键-值存储库往往具有横向的可扩展性，速度极快，或两者兼而有之。它们特别适合于数据相关性不高的问题。例如，在 Web 应用中，用户的会话数据满足这个标准，每个用户的会话活动会有所不同，并且大部分是与其他用户的活动无关的。

2. 不那么适合

往往缺乏索引和扫描功能，如果你需要能够执行数据查询，除了基本的 CRUD 操作（创建、读取、更新、删除）以外，KV 存储库的帮助不大。

9.1.3 列型

列型数据库（又称面向列的数据库，或列系列）与 KV 和 RDBMS 存储有许多的相似之处。像键-值存储库一样，值的查询通过匹配键完成。类似于关系数据库，把它们的值分组为零或多列，但是每一行可以填充任意多的数据。不同于前两个数据库，列型数据库按列存储类似的数据，而不是按行存储数据。列的添加很容易，版本控制是小菜一碟，并且对于空值没有存储成本。我们看到了 HBase 是对这一类型的经典实现。

1. 适合

传统上，横向扩展作为列型数据库开发的一个主要的设计目标。正因为如此，它们特别适合于在几十、几百或几千个节点的集群上的“大数据”问题。它们也往往内置支持如压缩和版本控制的功能。一个良好的列型数据存储问题的典型例子是索引网页。网页上有大量的文本（好处来自于压缩），在某种程度上相互关联，并随着时间变化（好处来自于版本控制）。

2. 不那么适合

不同的列型数据库有不同的特点，并由此带来不同缺点。但有一件事它们是相同的，那就是，最好基于你打算如何查询数据，设计你的数据库模式。这意味着，你应该预先对如何使用数据而不仅仅是数据将如何组成有一些想法。所以，如果你不能提前定义数据的使用模式（例如，需要快速的自由定义的报表），那么列型数据库未必是最合适的。

9.1.4 文档型

文档型数据库允许每个对象有任意数量的字段，甚至允许对象作为值以任意深度嵌套到其他字段中。这些对象通常用 JavaScript 对象符号（JSON）表示，MongoDB 和 CouchDB 都是这样，但这绝不是一个概念要求。由于文档型数据库不像关系数据库那样彼此相关，它们比较容易在几个服务器上实现分片和复制，这使得分布式实现相当普遍。MongoHQ 倾向于支持建立数据中心，管理 Web 上庞大的数据集，从而解决可用性问题。而 CouchDB 则侧重于简单耐用，可用性是通过相当自治的节点的主-主复制得到的。这些项目之间有很高的重叠性。

1. 适合

文档数据库适合于涉及高度可变领域的问题。当你事先不知道你的数据看起来究竟像什么样子，文档型数据库是一个不错的选择。此外，由于文档型数据库的性质，它们往往能很好地映射到面向对象编程模型。这意味着在数据库模型和应用模型之间移动数据时，阻抗性不匹配的情况较少。

2. 不那么适合

如果你习惯于在高度规范化的关系数据库模式中执行复杂的联接查询，你会发现文档型数据库的功能匮乏。文档型数据库一般应包含大部分或全部正常使用所需的有关信息。因此，在一个关系数据库中，你最好自然地规范化你的数据，以减少或消除可能不同步的副本，而使用文档型数据库，非规范化的数据是常态。

9.1.5　图

图数据库是一个新兴的数据库类型，更侧重于自由解释数据之间的相互关系而不是实际的数据值。作为我们的开源示例，Neo4j 在许多社交网络应用中日益普及。不像其他数据库类型将相似的对象划为共同的组，图数据库在形式上更自由——查询包含两个节点共享的边，即在节点之间移动。随着越来越多的项目使用它们，图数据库在简单的社交例子上不断发展，用于更多差异细微的使用场景，例如，推荐引擎、访问控制列表和地理数据。

1. 适合

图数据库似乎是为网络应用量身定做的。典型的例子是社交网络，其中节点代表相互之间有各种关系的用户。使用任何其他的类型对这种数据建模，往往难以适应，但图数据库会欣然接受。它们还是面向对象系统的完美匹配。如果可以在白板上建模数据，就可以在图中建模。

2. 不那么适合

由于节点之间的高度相互关联，因此图数据库一般不适合网络分区。因为图的快速爬取意味着你不能与其他数据库节点的网络联接，所以图数据库不能很好地向外扩展。可能的情况是，如果你使用图数据库，它会是一个较大系统的一部分，大容量数据存储在其他地方，而在图中只保存关系。

9.2 选择

正如我们在开始时所说的，数据是新的石油。我们坐在一片数据的汪洋大海之上，它是不可用的（一个更露骨的比喻是，现在数据中有很多钱），直到把它提炼为信息。轻松收集并最终存储、挖掘、提炼数据，从你选择的数据库开始。

对于给定的领域数据，决定选择哪个数据库往往比仅仅考虑哪个数据库类型最适合更为复杂。虽然社交图谱似乎显然用图数据库最好，但如果你希望用于 Facebook，数据太多，所以不能只选择一个。你更可能会选择“大数据”的实现，如 HBase 或 Riak。这将迫使你选择一个列型数据库或键-值存储库。在其他情况下，虽然你可能会认为关系数据库显然是银行交易的最好选项，但要了解的是，Neo4j 也支持 ACID 事务，这样就扩大你的选择范围。

这些例子足以指出，在选择哪个数据库或哪些数据库最好地服务于你的问题领域时，要考虑的不仅是数据库的类型。作为一般规则，当数据的大小增加，特定数据库类型的容量减小了。面向列的数据存储实现往往用于建立为跨数据中心，并支持最大的“大数据”集合，而图一般支持最小的集合。但是，情况并非总是如此。Riak 是一个大型的键-值存储库，用于跨越成百上千个节点的分片数据，而 Redis 的设计目标是运行在一个节点上，可能有几个主从复制或客户端管理的分片。

在选择数据库时，还有几个方面需要考虑，如耐用性、可用性、一致性、可扩展性和安全性等。你必须决定自由定义的查询是否重要，是否 mapreduce 就够了。你是否倾向于使用 HTTP/REST 接口，或者你是否愿意需要一个自定义的二进制协议的驱动程序？甚至更小范围的关注可能对你也很重要，如是否存在大量数据加载工具。

为了简化这些数据库之间的比较，我们创建了一个表，即附录 A。该表并不是一个详尽的功能列表。相反，它是一个工具，迅速比较那些我们已经介绍过的数据库。请注意每个数据库的版本。这些功能在眨眼间就会变化，所以我们强烈建议对于更新的版本，要慎重检查这些值。

9.3 我们将走向哪里

现代应用的伸缩性问题，现在主要是在数据管理的领域。我们已经达到了应用程序演化的一个点，其中编程语言、框架和操作系统的选择正变得非常便宜和容易，甚至是

硬件和运营业务（多方虚拟主机和“云”）也是如此。这些选择基本上变成了小问题，既是必须，也是某种偏好。如果你想在这个时代让你的应用可伸缩，就应该考虑选择哪个数据库或哪些数据库，它很可能是你真正的瓶颈。本书的主要目的，就是帮助你正确地做出这种选择。

虽然本书即将结束，但我们相信它已激起了你对多持久并存的兴趣。接下来是仔细研究引起你兴趣的数据库，或者继续学习其他选择，如 Cassandra、Drizzle 或 OrientDB。

开始动手吧。

附录 A

数据库概述表

本书包含了 7 种数据库的大量信息：PostgreSQL、Riak、HBase、MongoDB、CouchDB、Neo4j 和 Redis。接下来，你会看到两个表，记录了这些数据的一些方面，展示了本书详细介绍的内容的概要。虽然这些表不能替代真正的理解，但能提供一种粗略的感觉，让你了解各种数据库的优势和不足，以及在现代数据库全景图中的位置。

	MongoDB	CouchDB	Riak	Redis	PostgreSQL	Neo4j	HBase
类型	文档	文档	键-值	键-值	关系	图	列
版本	2.0	1.1	1.0	2.4	9.1	1.7	0.90.3
数据类型	有类型	有类型	Blob	半类型	预定义且有类型	无类型	预定义且有类型
数据关系	无	无	自由定义的（链接）	无	预定义	自由定义的（边）	无
标准对象	JSON	JSON	文本	字符串	表	哈希	列
编写语言	C++	Erlang	Erlang	C/C++	C	Java	Java
接口协议	TCP 上自定义	HTTP	HTTP、protbuf	TCP 上的简单文本	TCP 上自定义	HTTP	Thrift、HTTP
HTTP/REST	简单	是	是	否	否	是	是
二级索引	是	是	是	否	是	是（通过 Lucene）	是
版本化	否	是	是	否	否	否	是
Bulk Load	mongoimport	Bulk Doc API	否	否	COPY 命令	否	否

续表

	MongoDB	CouchDB	Riak	Redis	PostgreSQL	Neo4j	HBase
非常大的文件	GridFS	附件	Luwak（已弃用）	无	BLOB	无	无
自由定义的查询	命令、mapereduce	临时视图	弱支持、Lucene	命令	SQL	图遍历，Cypher、搜索	弱
mapreduce	JavaScript	JavaScript	JavaScript、Erlang	无	无	无（在分布式的情况下）	Hadoop
伸缩性	数据中心	数据中心（通过 BigCouch）	数据中心	集群（通过主-从）	集群（通过一些扩展）	集群（通过 HA）	数据中心
耐久性	写前日志，安全模式	只在崩溃时	写入法定数则耐久	仅追加的日志	ACID	ACID	写前日志
请求压缩	否	文件重写	否	快照	否	否	否
复制	主-从（通过复制集）	主-主	基于对等、主-主	主-从	主-从	主-从（在企业版中）	主-从
分片	是	是（通过 BigCouch 中的过滤器）	是	插件（如客户端）	扩展（如 PL/Proxy）	否	是，通过 HDFS
并发	写锁	无锁的 MVCC	向量锁	无	表/行写锁	写锁	每行一致
事务	无	无	无	多操作队列	ACID	ACID	是（在打开时）
触发器	无	更新验证或改变 API	提交前后	无	是	事务事件处理程序	无
安全性	用户	用户	无	口令	用户/群	无	Kerberos，通过 Hadoop 的安全性
一主机多实例	是	是	否	否	是	否	否
主要区别	容易查询大数据	耐久的、可嵌入的集群	高可用性	非常非常快	最好的 OSS RDBMS 模型	灵活的图	规模非常大，Hadoop 基础设施
不足	嵌入能力	查询能力	查询能力	复杂数据	分布式可用性	BLOB 或 TB 级数据	灵活增长，查询能力

附录 B

CAP 定理

理解 5 种数据库类型是一个重要的选择标准，但并不是唯一的标准。在本书中另一个反复出现的主题是 CAP 定理，它揭示了一个令人不安的事实，即面对网络的不稳定性，分布式数据库系统如何表现。

CAP 证明了，可以创建一个分布式数据库，它是一致的（写入是原子的而且所有后续请求检索出新的值），可用的（只要一台服务器在运行，数据库将始终返回值），或者分区容错的（即使服务器的通信暂时中断了，系统仍将运行——那就是，网络分区），但是你只能同时拥有以上特性中的两个。

换句话说，可以创建一个分布式数据库系统，它是一致的和分区容错的，系统是可用的和分区容错的，或系统是一致的和可用的（但不是分区容错的，这基本上意味着不是分布的）。但是不可能创建一个分布式数据库，它是一致的、可用的，同时也是分区容错的。

CAP 定理在考虑一个分布式数据库时是有用的，因为你必须决定你愿意放弃什么。你选择的数据库将失去可用性或一致性。分区容错是严格意义上的架构决策（数据库是否是分布的）。重要的是要了解 CAP 定理以充分把握你的选择。在本书中对数据库实现的取舍在很大程度上受 CAP 定理影响。

B.1　最终一致性

分布式数据库必须是分区容错的，所以在可用性和一致性之间的选择很困难。然而，CAP 指出，如果你选择了可用性，你就不能拥有真正的一致性，你仍然可以提供最终一致性。

CAP 冒险，第一部分：CAP

将世界想象为一个巨大的分布式数据库系统。在世界上的所有陆地上包含关于某些主题的信息，只要你在人或技术的附近，你可以找到你问题的答案。

现在，为了论证，想象你是碧昂丝·诺尔斯（Beyonce Knowles）的一个铁杆粉丝，而日期是 2006 年 9 月 5 日。突然，在你朋友的海滨别墅庆祝碧昂丝第二张录音专辑发行的聚会上，一个奇怪的波浪席卷了码头，并把你拖到了海里。你做了一个临时的筏子并在几天后被冲到了一个荒岛上。没有任何的通信手段，你实际上与系统（整个世界）的其他部分隔离开来。在那里你等了长长的五年……

在 2011 年的一个早晨，你被来自海上的喊声惊醒。一位富有经验的纵帆船老船长发现了你！在你单独一人五年之后，船长弯腰大声问道："碧昂丝有多少张录音专辑？"

你现在要作一个决定。你可以用你所拥有的最近（现在是五年前）的值回答这个问题。如果你回答他的问题，你是可用的。或者，你可以拒绝回答这个问题，因为你知道你被隔离了，你的回答可能无法与世界其他地方保持一致。船长不会得到回答，但世界的状态保持了一致（如果他驾船回家，他就可以得到正确的答案）。你作为被查询的节点，可以帮助保持世界数据的一致或可用，但不能同时两者兼备。

最终一致性背后的思想是，每个节点始终可用于服务请求。作为一个权衡，数据修改在后台被传播到其他节点。这意味着，在任何时候，系统可能会不一致，但是大体上数据仍是准确的。

互联网域名服务（Domain Name Service，DNS）是最终一致性的一个典型例子。你注册一个域名，可能需要几天时间传播到在互联网上的所有 DNS 服务器。但是没有任何时候任何特定的 DNS 服务器是不可用的（假设你可以连接到它，那它就是可用的）。

B.2 实际中的 CAP

一些分区容错的数据库可以调整为对每个请求或多或少一致或可用。Riak 的运行就像这样，允许客户端在请求时决定它们需要什么程度的一致性。本书中的其他数据库在很大程度上占据了 CAP 权衡三角形的某一个角。

CAP 冒险，第二部分：最终一致性

让我们回到两年前，2009 年。在这个时间点上，你已经在岛上待了三年，你在沙子中发现了一个瓶子，这是与外界的宝贵接触。你拔去瓶塞，欢欣鼓舞！你刚刚获得一项完整的知识……

> 碧昂丝录音专辑的数量对世界的总知识是至关重要的。它是如此重要，事实上，她每次发行新专辑，就有人在纸上写上当前日期和数量。他们把这张纸放到一个瓶子里，并把它扔到海里。如果有人，像你自己一样，在一个荒岛上与世隔绝，他们最终能得到正确的答案。
>
> 向前跳转到现在。当船长问你:“碧昂丝有多少张录音专辑？”你仍然可用并回答:“三张。”你可能与世界其他地方是不一致的，但在尚未收到另一个瓶子之前，你对你的答案相当肯定。
>
> 故事的结局是，船长救了你，你回到了家，发现了碧昂丝的新专辑，并从此过上了幸福的生活。只要你留在陆地上，你不需要分区容错，并能保持一致性和可用性，直到你的生命结束。

Redis、PostgreSQL 和 Neo4j 是一致的和可用的（CA）；它们不分布数据，因此分区不是一个问题（虽然可以说，CAP 在非分布式系统中并没有多大意义）。MongoDB 和 HBase 一般是一致的和分区容错的（CP）。在网络分区的情况下，它们可能无法回应特定类型的查询（例如，在 Mongo 副本中设置标记 `slaveok` 为 false，用于读取）。在实践中，完善地处理硬件故障——其他仍然联网的节点可以为宕机的服务器提供后备支持——但是严格说来，在 CAP 定理的意义中，它们是不可用的。最后，即使两个或更多的 CouchDB 服务器可以在它们之间复制数据，CouchDB 不保证任意两个服务器之间的一致性。

值得注意的是，这些数据库中的大部分可以通过配置改变其 CAP 类型（Mongo 可以是 CA，CouchDB 可以是 CP），但在这里，我们已经注意到它们的默认或通常行为。

B.3 延迟权衡

然而，分布式数据库系统的设计不仅仅需要CAP。例如，低延迟（速度）是许多架构师的主要关注。如果你读过亚马逊的Dynamo[1]论文，你会注意到很多关于可用性和亚马逊的延迟要求的讨论。对于特定类型的应用，即便是很小的延迟变化也可能转化为一个大的成本。著名的例子是，雅虎的PNUTS数据库放弃了正常运行的可用性和分区上的一致性，从而在设计上挤压出低延迟[2]。重要的是，在与分布式数据库打交道时要考虑CAP，但同样重要的是要知道，分布式数据库理论并不止于此。

---

1 http://allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf。
2 http://dbmsmusings.blogspot.com/2010/04/problems-with-cap-and-yahoos-little.html。

七周七数据库

Seven Databases in Seven Weeks

A Guide to Modern Databases and the NoSQL Movement

如今，我们要面对和使用的数据正在变得越来越庞大、越来越复杂。如果说数据是新的石油，那么数据库就是油田、炼油厂、钻井和油泵。作为一名现代的软件开发者，我们需要了解数据管理的新领域，既包括RDBMS，也包括NoSQL。

本书遵循《七周七语言》的写作风格和体例，带领你学习和了解当今最热门的开源数据库。

本书涉及如下7种数据库。它们分别代表了5种数据模型，而且就像是数据库世界里的7种工具和“武器”：

◎“锤子”PostgreSQL（关系型）；

◎“钢筋”Riak（键-值型）；

◎“射钉枪”HBase（列型）；

◎“电钻”MongoDB（文档型）；

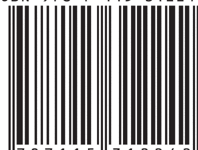
◎“扳手”CouchDB（文档型）；

◎“蹦极绳”Neo4j（图型）；

◎“润滑油”Redis（键-值型）。

本书不是简单介绍每种技术，而是探讨了每种技术的核心基本概念。本书将带你深入每种数据库，了解它们的长处与不足，了解为了满足你自己的需求如何做出选择。通过本书，你将掌握这7种武器，从而能够处理真实世界的问题，并且深刻了解哪种问题最适合哪种类型的工具。

ISBN 978-7-115-31224-2



封面设计：任文杰

分类建议：计算机／数据库

人民邮电出版社网址：www.ptpress.com.cn